VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Server pro podporu výuky teoretické informatiky

Supporting Server for Theoretical Computer Science Teaching

2016 Jan Lesniak

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Lesniak**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Server pro podporu výuky teoretické informatiky
Supporting Server for Theoretical Computer Science Teaching

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je vytvořit server, na kterém si studenti budou moci spustit výpočet vybraných algoritmů z oblasti teoretické informatiky na jimi zadaných i ukázkových vstupech a na zobrazeném průběhu výpočtu lépe pochopit princip fungování těchto algoritmů. Tato práce se konkrétně zaměří na algoritmy z oblasti výrokové logiky.

1. Naprogramujte řešení problémů z oblasti výrokové logiky. Konkrétně pro danou formuli konstrukci pravdivostní tabulky, provádění ekvivalentních úprav, převod formulí do normálních forem, rezoluční metodu.
2. Cílem není nejefektivnější implementace algoritmů, ale taková, která umožní zobrazovat postupně jednotlivé kroky tak, aby uživatel mohl pochopit princip jejich fungování.
3. Vytvořte výukový server, kde bude možné zadat vstupní data pro naprogamované algoritmy a pro tato data si nechat zobrazit postup výpočtu.
4. Vytvořte i ukázkové vstupy pro jednotlivé algoritmy, aby postup výpočtu bylo možné zobrazit i bez zadávání vstupu uživatelem.

Seznam doporučené odborné literatury:

[1] P. Jančar: Teoretická informatika (výukový text), FEI VŠB - TUO, 2007.

Další literatura dle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Kot, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016 .....

Rád bych na tomto místě poděkoval panu Ing. Martinu Kotovi, Ph.D., za odborné vedení a cenné rady, které mi poskytoval během tvorby mé diplomové práce.

Abstrakt

Tato práce se zabývá implementací a problematikou algoritmů pro zprostředkování postupů a výsledků úloh používaných při výuce výrokové logiky, kterými jsou tabulková metoda, rezoluční metoda a logicky ekvivalentní úpravy pro převod formule do normálních forem. Popisuje a vysvětluje algoritmy pro zpracování uživatelského vstupu a pro každý typ úlohy. Výsledkem je webová aplikace dostupná z internetu, která uživateli poskytne automatizovaný postup při řešení těchto problémů.

Klíčová slova: Výroková logika, tabulková metoda, rezoluční metoda, ekvivalentní úpravy formulí, lexikální analýza, syntaktická analýza

Abstract

This thesis deals with algorithms for visualization of procedures and results of propositional's logic tasks in education, which are table method, resolution method and logically equivalent edits for transformation of formulas to normal forms. There is the description and explanation of used algorithms for user's input and each type of task. Web application with automatic visualization of the gradual approach to task results is the result of this thesis.

Key Words: Propositional logic, table method, resolution method, logically equivalent formulas, lexical analysis, syntactic analysis

Obsah

Seznam použitých zkratek a symbolů

CSS	–	Cascading Style Sheets
C#	–	C Sharp
DNF	–	Disjunktivní Normální Forma
GUI	–	Graphical User Interface
HTML	–	Hyper Text Markup Language
KNF	–	Konjunktivní Normální Forma
SA	–	Syntaktický Analyzátor
ÚDNF	–	Úplná Disjunktivní Normální Forma
ÚKNF	–	Úplná Konjunktivní Normální Forma

Seznam obrázků

Seznam tabulek

1 Úvod

Logické myšlení a vyjadřování je součástí našeho podvědomí, aniž bychom znali zákony logiky. Dokud se lidé nezačali do větší hloubky zabývat skladbou vět a její strukturou, neuvědomovali jsme si, že používáme gramatická pravidla, jimiž se používání jazyka řídí. Jelikož je jazyk používán i ve vztahu mezi člověkem a strojem, musí být nějakým způsobem formalizován. Nejjednodušší formou logiky je logika 0-tého řádu nebo-li výroková logika, ve které primitivní formule (výrokové proměnné) nemají žádnou vnitřní stavbu a jediným jejich atributem je pravdivostní hodnota. Existují další složitější logiky vyšších řádů, které zahrnují vlastnosti individuí, vztahy mezi nimi a vztahy mezi vztahy těchto individuí, ale těmto se věnovat nebudeme [2].

Tato práce se zabývá výrokovou logikou, konkrétněji pak využitím moderních technologií k vytvoření webové aplikace, která bude sloužit jako pomůcka pro výuku části teoretické informatiky a k lepšímu pochopení některých disciplín problematiky výrokové logiky. Aplikace umí nejen ukázat konečné výsledky různých problémů, ale navíc popisuje, jak se k těmto výsledkům postupně dostat a zajistit tak, aby uživatel danému řešení problému porozuměl. Aplikace konkrétně umí ze zadané formule vytvořit výstup tabulkové metody a určit zda je formule splnitelná, tautologie, nebo kontradikce, dále umí názorně předvést převody pomocí ekvivalentních úprav do ÚDNF, DNF, ÚKNF, KNF a také umí demonstrovat rezoluční metodu. Aplikace je dostupná z internetu na webové adrese vl-vsb.aspone.cz.

V následující kapitole 2 je podrobněji rozebrána výroková logika a všechny její náležitosti vztahující se k této práci. Další kapitoly se věnují implementaci. V kapitole 3 popisuji návrh implementace a co vše je třeba k fungování aplikace. Jde o obecnou představu, jak aplikace vnitřně funguje. Kapitola 4 popisuje algoritmy ke zpracování vstupních formulí zadaných uživatelem. V kapitole 5 jsou detailně rozebrány algoritmy použité k řešení problému pravdivostní tabulky, ekvivalentních úprav a rezoluční metody. A v poslední kapitole 6 naleznete závěrečné zhodnocení práce, její úskalí a návrhy na vylepšení aplikace.

2 Výroková logika

Tato kapitola obsahuje teorii a definování pojmů výrokové logiky, které budeme potřebovat v dalších kapitolách pojednávajících o konkrétních řešeních problémů jako je tabulková metoda, rezoluční metoda, ekvivalentní úpravy a převody do normálních forem.

2.1 Definice důležitých pojmů

Výroková logika umožňuje analyzovat věty pouze do úrovně elementárních výroků, jejichž strukturu již dále nezkoumá. Výrok je tvrzení, o němž má smysl prohlásit, zda je pravdivé či nepravdivé. Z těchto dvou definic ale zjistíme, že ne každá věta vyjadřuje výrok. Mohou existovat tvrzení, o kterých nemůžeme říci, zda jsou pravdivé nebo nepravdivé. Například věta: „Teplota vzduchu v Ostravě.“, nebude výrok.

Výroky dělíme na jednoduché a složené. Elementární (jednoduchý) výrok je takové tvrzení, jehož žádná vlastní část již není výrokem. Složený výrok pak má vlastní části – výroky. Výroková logika zkoumá strukturu těchto složených výroků v tom smyslu, že zkoumá způsob skládání jednoduchých výroků do složených pomocí logických spojek. Je to tedy teorie logických spojek. Přitom ovšem zachovává žádoucí princip skladebnosti (kompozicionality), podle něhož je pravdivostní hodnota složeného výroku jednoznačně určena jen pravdivostními hodnotami jeho složek a povahou spojení těchto složek (tj. logickou povahou spojek). Příklad složeného výroku je: „V Ostravě prší a v Brně svítí slunce.“

Jazyk výrokové logiky musí proto obsahovat symboly zastupující jednotlivé elementární výroky, tzv. výrokové symboly (proměnné), které budou nabývat hodnot pravda, nepravda, symboly pro logické spojky a případné pomocné symboly. Každý jazyk je nejlépe definován abecedou a gramatikou. Abeceda určuje typy symbolů, které mohou být v jazyce použity, gramatika pak zadává pravidla tvorby (nekonečně mnoha) dobře utvořených výrazů, tj. v tomto případě formulí. [2]

Abeceda jazyka výrokové logiky je množina následujících symbolů:

- Výrokové symboly: $p, q, r, ...$
- Symboly logických spojek: $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$
  Symboly nazýváme po řadě spojky negace, disjunkce, konjunkce, implikace, ekvivalence.
- Závorky: $(,)$

Gramatika jazyka výrokové logiky rekurzivně definuje nekonečnou množinu formulí:

1. Výrokové symboly jsou formule.
2. Jsou-li výrazy A, B formule, pak jsou formulemi i výrazy

$$(\neg A), (A \vee B), (A \wedge B), (A \Rightarrow B), (A \Leftrightarrow B)$$

3. Jiných formulí výrokové logiky, než podle bodů 1. a 2. není.

Výrokové symboly elementárních formulí budu v následujícím textu zapisovat pomocí malých písmen abecedy. Velká písmena označují formule složené.

Symboly logických spojek se mohou v různých publikacích lišit, ale měly by se napříč textem dodržovat stejné. Budu tedy využívat spojek výše zmíněných.

Pravdivostní ohodnocení (valuace) výrokových symbolů, je zobrazení v, které ke každému výrokovému symbolu přiřazuje pravdivostní hodnotu, tj. hodnotu z množiny $\{1, 0\}$, která kóduje množinu $\{pravda, nepravda\}$.

Pravdivostní funkce formule výrokové logiky je funkce w, která ke každému pravdivostnímu ohodnocení výrokových symbolů přiřazuje pravdivostní hodnotu celé formule. Tato honota je určena takto:

1. Pravdivostní hodnota elementární formule je rovna valuaci výrokového symbolu, tj. $w(p)_v = v(p)$ pro všechny výrokové proměnné p.

2. Jsou-li dány pravdivostní funkce formulí A, B, pak pravdivostní funkce formulí $\neg A$, $A \vee B$, $A \wedge B$, $A \Rightarrow B$, $A \Leftrightarrow B$ jsou dány následující tabulkou:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

Tabulka 1: Pravdivostní funkce

Protože cílem této práce není převod z přirozeného jazyka do výrokových formulí, ale pracuji už rovnou se zadanými formulemi, uvedu jen zběžně a pro úplnost příklady souvislosti mezi výrokovými spojkami a přirozeným jazykem.

Analýza na základě výrokové logiky nám umožňuje studovat strukturu vět z hlediska skládání jednoduchých výroků do složených výroků pomocí logických spojek. Elementární výroky zde považujeme za nestrukturované "cihly", které skládáme do strukturovaných bloků. Elementární výroky vstupují do spojení jen svou pravdivostní hodnotou a jsou navzájem zcela nezávislé. [2]

- Spojka **negace**, značíme $\neg$, odpovídá **„Není pravda, že“**
  Je to unární spojka, nespojuje dva výroky.
  Příklad: „Není pravda, že Praha je velkoměsto“ zapíšeme $\neg p$

- Spojka **konjunkce**, značíme $\wedge$, odpovídá **„a“**
  Je to binární, komutativní spojka.
  Příklad: „Praha je hlavní město ČR a Praha je sídlo prezidenta ČR“ zapíšeme $p \wedge q$

- Spojka **disjunkce**, značíme $\vee$, odpovídá **„nebo“**
  Je to binární, komutativní spojka.
  Příklad: „Osobní auta mají přední nebo zadní náhon“ zapíšeme $p \vee q$

- Spojka **implikace**, značíme $\Rightarrow$, odpovídá **„jestliže, pak“**, **„když, tak“**, **„je-li, pak“**, apod.
  Je to binární spojka, která není komutativní.
  Příklad: „Jestliže existují ufoni, tak jsem papež“ zapíšeme $p \Rightarrow q$

- Spojka **ekvivalence**, značíme $\Leftrightarrow$, odpovídá **„právě tehdy, když“**, **„tehdy a jen tehdy, když“**, apod.
  Je to binární spojka, která není komutativní.
  Příklad: „Dám ti dobré hodnocení, tehdy a jen tehdy, když budeš umět“ zapíšeme $p \Leftrightarrow q$ [2]

Další důležitou věcí jsou priority logických spojek. Aby zapisované formule výrokové logiky vypadaly přehledně a čistě můžeme přijmout konvenci o omezení používání závorek, která vypadá následovně:

- Složenou formuli nejvyššího řádu netřeba závorkovat.

- Logické spojky uspořádáme do prioritní stupnice $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. Ze dvou spojek váže silněji ta, která je v uvedené stupnici umístěna nejvíce vlevo. Není ale dobré tuto konvenci zneužívat a vytvářet tak spíše nepřehledné formule. V některých případech je pro přehlednost lepší použití závorek a vyznačit tak strukturu formule.

- V případě, že o prioritě vyhodnocení nerozhodnou ani závorky ani prioritní stupnice, vyhodnocujeme formuli zleva doprava. Např. formuli $a \Rightarrow b \Rightarrow c \Rightarrow d$ vyhodnocujeme tak, jakoby byla zapsána ve tvaru $((a \Rightarrow b) \Rightarrow c) \Rightarrow d$.

- U vícečlenných konjunkcí nebo disjunkcí není třeba (vzhledem k jejich asociativitě) uvádět závorky, tj. např. místo $(a \wedge b) \wedge c$ nebo $a \wedge (b \wedge c)$ lze psát $a \wedge b \wedge c$. Tato konvence souvisí s předchozí konvencí (na pořadí vyhodnocování nezáleží a tedy lze standardně vyhodnocovat zleva doprava).[2]

2.2 Tabulková metoda

Nyní se podíváme na nejjednodušší metodu dokazování ve výrokové logice tabulkovou metodu. Touto metodou sémanticky dokazujeme, zda je zadaná formule splnitelná, tautologie, nebo kontradikce. Také určuje, co jsou modely zadané formule.

Každé ohodnocení v výrokových symbolů obsažených ve formuli A, pro které je hodnota pravdivostní funkce rovna 1, tedy $w(A)_v = 1$, se nazývá **model** této formule.

Formule A výrokové logiky je **splnitelná**, je-li $w(A)_v = 1$ pro nějaké ohodnocení v, neboli existuje aspoň jeden model formule A.

Formule A výrokové logiky je **tautologií**, je-li $w(A)_v = 1$ pro všechna ohodnocení v, neboli každé ohodnocení je modelem formule A. Skutečnost, že formule A je tautologií, označujeme zápisem $\models A$.

Formule A výrokové logiky je **kontradikcí**, jestliže neexistuje takové ohodnocení výrokových symbolů, pro které by hodnota pravdivostní funkce formule A byla rovna 1, tj. $w(A)_v = 0$ pro všechna ohodnocení v, formule nemá model. [2]

Příklad tabulky pro formuli definovanou jako $\neg((\neg p \vee \neg q) \Leftrightarrow \neg(p \wedge q))$ může vypadat třeba tak, jako v Tabulce 2.

$\neg$	$((\neg$	p	$\vee$	$\neg$	$q)$	$\Leftrightarrow$	$\neg$	$(p$	$\wedge$	$q))$
4	1	0	2	1	0	3	2	0	1	0
0	1	0	1	1	0	1	1	0	0	0
0	1	0	1	0	1	1	1	0	0	1
0	0	1	1	1	0	1	1	1	0	0
0	0	1	0	0	1	1	0	1	1	1

Tabulka 2: Příklad tabulkové metody 1

První řádek Tabulky 2 obsahuje zadanou formuli, která je rozdělená na jednotlivé logické spojky a výrokové symboly. Na druhém řádku jsou čísla určující pořadí zpracování logických spojek, kde nejnižší číslo má nejvyšší prioritu ve smyslu nejdřívějšího zpracování. Výrokové symboly mají vždy největší prioritu, proto dostanou číslo 0. Další řádky jsou pak pravdivostními ohodnoceními pro jednotlivé kombinace výrokových symbolů. Počet řádků tabulky exponenciálně roste s počtem výrokových symbolů.

Existuje i další způsob zápisu tabulkové metody, jak nám ukazuje Tabulka 3. Můžeme vidět, že se liší jednak kompaktností a přehledností, ale i způsobem vyplňování. V této druhé tabulce lze lépe přečíst, které hodnoty výrokových symbolů jsou modely formule a které ne. Posloupnost zpracování logických spojek je dána zápisem postupně narůstajících formulí zleva doprava. Úplně vlevo jsou jednotlivé výrokové symboly a úplně vpravo pak celá formule. Jednotlivé řádky, stejně jako v předchozí metodě, obsahují pravdivostní funkce výrokových formulí.

p	q	$\neg p$	$\neg q$	$(p \wedge q)$	$(\neg p \wedge \neg q)$	$\neg(p \wedge q)$	$((\neg p \wedge \neg q) \Leftrightarrow \neg(p \wedge q))$	$\neg((\neg p \wedge \neg q) \Leftrightarrow \neg(p \wedge q))$
0	0	1	1	0	1	1	1	0
0	1	1	0	0	1	1	1	0
1	0	0	1	0	1	1	1	0
1	1	0	0	1	0	0	1	0

Tabulka 3: Příklad tabulkové metody 2

2.3 Ekvivalentní úpravy

Ekvivalentními úpravami ve výrokové logice rozumíme takové úpravy výrokových formulí, které změní podobu zadaných formulí, ale nezmění jejich pravdivostní funkci. Pomocí takových úprav můžeme formule převádět do forem, které následně slouží k různým účelům. Nejpoužívanější formou je konjunktivní normální forma **KNF**, která má velké uplatnění v číslicové technice, například ke tvorbě Karnaughových map [1]. Využívá pouze booleovských spojek jednoduše realizovatelných jako hradla v číslicových obvodech. Také je tato forma základem pro rezoluční metodu. Pro ekvivalentní úpravy můžeme využít jakoukoliv platnou ekvivalenci mezi formulemi (nebo také jakoukoliv tautologii ve tvaru implikace). Bývá ale zvykem používat jen několik jednoduchých ekvivalencí, ty ostatní by bylo možné docílit sekvencí několika kroků používajících ony jednoduché ekvivalence. Následuje výpis těch nejdůležitějších z nich a těch, které jsou použity v aplikaci.

- Zákon vyloučení třetího: $\models \mathbf{p} \vee \neg\mathbf{p}$
- Zákon dvojí negace: $\models \mathbf{p} \Leftrightarrow \neg\neg\mathbf{p}$
- Jsou-li 1 a 0 atomické formule s významem 1 = konstanta Pravda a 0 = Nepravda, pak dále platí:
  $\models \mathbf{p} \vee \mathbf{1}$
  $\models (\mathbf{p} \vee \mathbf{0}) \Leftrightarrow \mathbf{p}$
  $\models (\mathbf{p} \wedge \mathbf{1}) \Leftrightarrow \mathbf{p}$
- Komutativní zákon pro $\vee$: $\models (\mathbf{p} \vee \mathbf{q}) \Leftrightarrow (\mathbf{q} \vee \mathbf{p})$
- Komutativní zákon pro $\wedge$: $\models (\mathbf{p} \wedge \mathbf{q}) \Leftrightarrow (\mathbf{q} \wedge \mathbf{p})$
- Komutativní zákon pro $\Leftrightarrow$: $\models (\mathbf{p} \Leftrightarrow \mathbf{q}) \Leftrightarrow (\mathbf{q} \Leftrightarrow \mathbf{p})$
- Asociativní zákon pro $\vee$: $\models (\mathbf{p} \vee \mathbf{q}) \vee \mathbf{r} \Leftrightarrow \mathbf{p} \vee (\mathbf{q} \vee \mathbf{r})$
- Asociativní zákon pro $\wedge$: $\models (\mathbf{p} \wedge \mathbf{q}) \wedge \mathbf{r} \Leftrightarrow \mathbf{p} \wedge (\mathbf{q} \wedge \mathbf{r})$
- Asociativní zákon pro $\Leftrightarrow$: $\models (\mathbf{p} \Leftrightarrow \mathbf{q}) \Leftrightarrow \mathbf{r} \Leftrightarrow \mathbf{p} \Leftrightarrow (\mathbf{q} \Leftrightarrow \mathbf{r})$
- Distributivní zákon pro $\wedge, \vee$: $\models (\mathbf{p} \vee \mathbf{q}) \wedge \mathbf{r} \Leftrightarrow (\mathbf{p} \wedge \mathbf{r}) \vee (\mathbf{q} \wedge \mathbf{r})$
- Distributivní zákon pro $\vee, \wedge$: $\models (\mathbf{p} \wedge \mathbf{q}) \vee \mathbf{r} \Leftrightarrow (\mathbf{p} \vee \mathbf{r}) \wedge (\mathbf{q} \vee \mathbf{r})$
- Zákon odstranění ekvivalence: $(\mathbf{p} \Leftrightarrow \mathbf{q}) \Leftrightarrow (\mathbf{p} \Rightarrow \mathbf{q}) \wedge (\mathbf{q} \Rightarrow \mathbf{p})$
- Zákon odstranění implikace: $(\mathbf{p} \Rightarrow \mathbf{q}) \Leftrightarrow (\neg\mathbf{p} \vee \mathbf{q})$
- De Morganovy zákony:
  $\neg(\mathbf{p} \wedge \mathbf{q}) \Leftrightarrow (\mathbf{p} \vee \mathbf{q})$
  $\neg(\mathbf{p} \vee \mathbf{q}) \Leftrightarrow (\mathbf{p} \wedge \mathbf{q})$

- Zákon identity: $\mathbf{p} \Leftrightarrow \mathbf{p}$
- Zákon idempotence $\vee$: $(\mathbf{p} \vee \mathbf{p}) \Leftrightarrow \mathbf{p}$
- Zákon idempotence $\wedge$: $(\mathbf{p} \wedge \mathbf{p}) \Leftrightarrow \mathbf{p}$
- Zákon sporu: $(\mathbf{p} \wedge \neg\mathbf{p}) \Leftrightarrow \mathbf{0}$
- Agresivnost 0/1:
  $(\mathbf{p} \vee \mathbf{1}) \Leftrightarrow \mathbf{1}$
  $(\mathbf{p} \wedge \mathbf{0}) \Leftrightarrow \mathbf{0}$
- Zákon rozšíření
  $\mathbf{p} \Leftrightarrow ((\mathbf{p} \wedge \mathbf{q}) \vee (\mathbf{p} \wedge \neg\mathbf{q}))$
  $\mathbf{p} \Leftrightarrow ((\mathbf{p} \vee \mathbf{q}) \wedge (\mathbf{p} \vee \neg\mathbf{q}))$

[2] [4]

Každá formule má svou pravdivostní funkci. Tu stejnou funkci ale mohou sdílet různé jiné formule, které jsou navzájem ekvivalentní. Abychom tuto nejednoznačnost odstranili, definujme standardní tvary formulí výrokové logiky. Každá třída navzájem ekvivalentních formulí bude reprezentována jedinou formulí ve standardním tvaru a to ve tvaru ÚDNF nebo ÚKNF. Následují definice těchto a dalších forem formulí.

- **Literál** je výroková proměnná (tj. atomická formule) nebo její negace.
- **Elementární konjunkce (EK)** je konjunkce literálů.
- **Elementární disjunkce (ED)** je disjunkce literálů.
- **Úplná elementární konjunkce (ÚEK)** dané množiny výrokových proměnných je elementární konjunkce, ve které se každá proměnná z dané množiny vyskytuje právě jednou (buďto prostě nebo negovaná).
- **Úplná elementární disjunkce (ÚED)** dané množiny výrokových proměnných je elementární disjunkce, ve které se každá proměnná z dané množiny vyskytuje právě jednou (buďto prostě nebo negovaná).
- **Disjunktivní normální forma (DNF)** dané formule je formule ekvivalentní s danou formulí a mající tvar disjunkce elementárních konjunkcí.
- **Konjunktivní normální forma (KNF)** dané formule je formule ekvivalentní s danou formulí a mající tvar konjunkce elementárních disjunkcí.
- **Úplná disjunktivní normální forma (ÚDNF)** dané formule je formule ekvivalentní s danou formulí a mající tvar disjunkce úplných elementárních konjunkcí.

- **Úplná konjunktivní normální forma (ÚKNF)** dané formule je formule ekvivalentní s danou formulí a mající tvar konjunkce úplných elementárních disjunkcí. [2]

Z důvodu takto definované struktury ÚDNF a DNF není možné těmito formami zapsat formuli, která je kontradikcí. Podobně je tomu u ÚKNF a KNF, kterými není možné zapsat formuli, která je tautologií.

Příklad převodu formule $(a \Rightarrow b) \wedge (a \Rightarrow b)$ do ÚNDF pomocí ekvivalentních úprav:

1. $(a \Rightarrow b) \wedge (a \Rightarrow b)$
2. $(\neg a \vee b) \wedge (a \Rightarrow b)$
3. $(\neg a \vee b) \wedge (\neg a \vee b)$
4. $(\neg a \wedge \neg a) \vee (b \wedge \neg a) \vee (\neg a \wedge b) \vee (b \wedge b)$
5. $\neg a \vee (\neg a \wedge b) \vee b$
6. $(\neg a \wedge b) \vee (\neg a \wedge \neg b) \vee (b \wedge \neg a) \vee (\neg a \wedge b) \vee (b \wedge a) \vee (b \wedge \neg a)$
7. $(\neg a \wedge b) \vee (\neg a \wedge \neg b) \vee (b \wedge a)$

2.4 Rezoluční metoda

Tato metoda ve výrokové logice slouží k dokazování nesplnitelnosti dané formule (resp. množiny klauzulí). Klauzule jsou formule zapsané v KNF. Princip je v hledání sporu pomocí jediného rezolučního pravidla, které může být opakovaně použito na zadané klauzule. Můžeme tuto metodu využít i k ověření tautologičnosti formule, nebo zjištění správnosti úsudku. Protože rezoluční metoda, pokud je množina klauzulí sporná, vždy najde spor, je vhodná pro strojové dokazování a používá se v různých oblastech umělé inteligence. Byla zavedena Alanem Robinsonem v roce 1965 [3].

Rezoluční pravidlo odvozování: Nechť l je literál. Z formule $(A \vee l) \wedge (B \vee \neg l)$ odvoď formuli $(A \vee B)$. Zapisujeme:

$$\frac{(A \vee l) \wedge (B \vee \neg l)}{(A \vee B)}$$

Toto pravidlo není přechodem k ekvivalentní formuli, ale zachovává pravdivost, tedy rezolvent z daných předpokladů vyplývá. Rezolvent je to, co vznikne po použití rezolučního pravidla.

Rezoluční metoda se opírá o tyto tři principy:

- **Princip vyvrácení**, převádějící problém důkazu dané formule na problém důkazu nesplnitelnosti negace této formule.
- **Rezoluční odvozovací pravidlo** je jediné používané odvozovací pravidlo.
- **Robinsonův rezoluční princip** umožňující vyvodit spor z negované původní formule a tak dokázat její nesplnitelnost a tím dokázat platnost původní formule.

Příklady použití rezoluční metody:

- Důkaz, že množina klauzulí je nesplnitelná:

  - Na množinu klauzulí uplatňujeme rezoluční pravidlo.
  - Pokud při uplatňování rezolučního pravidla dospějeme k prázdné klauzuli, je tato množina nesplnitelná.

- Důkaz, že formule A je tautologie:
  - Formuli A znegujeme a převedeme do KNF.
  - Uplatňujeme rezoluční pravidlo.
  - Pokud při uplatňování rezolučního pravidla dospějeme k prázdné klauzuli, je negovaná formule $\neg A$ nesplnitelná a původní formule A je tautologie.

- Důkaz správnosti úsudku $P_1, ..., P_n \models Z$:
  - Závěr Z znegujeme a dokazujeme, že množina $\{P_1, ..., P_n, \neg Z\}$ je sporná.
  - Jinými slovy dokazujeme, že formule $(P_1 \wedge P_2 \wedge ... \wedge P_n) \Rightarrow Z$ je tautologie , tedy že její negace $P_1 \wedge P_2 \wedge ... \wedge P_n \wedge \neg Z$ je kontradikce.

Konkrétní příklad ověření platnosti úsudku: $(p \vee q), (p \Rightarrow r), (\neg q \vee s) \models (r \vee s)$
Negujeme závěr a převedeme na klauzule:

1.	$p \vee q$	klauzule
2.	$\neg p \vee r$	klauzule
3.	$\neg q \vee s$	klauzule
4.	$\neg r$	negovaný závěr
5.	$\neg s$	negovaný závěr
6.	$\neg q$	rezolvent 3. 5.
7.	$\neg p$	rezolvent 2. 4.
8.	$p \vee s$	rezolvent 1. 3.
9.	p	rezolvent 5. 8.
10.	$\square$	prázdná klauzule 7. 9.

3 Popis implementace

Program je vytvořený jako webová aplikace na platformě ASP.NET MVC společnosti Microsoft. Skládá se z klientské části, kde je uživateli umožněna komunikace se serverem pomocí grafického uživatelského rozhraní, dále jen „GUI“ a serverové části, která se stará o všechny výpočty a zasílá jejich výsledky zpět uživateli. Aplikace používá několik technologií:

- **ASP.NET MVC**
  Celá aplikace je postavena na této technologii. Zahrnuje jak klientskou část, tak serverovou. Používá architektonický styl *model view controller*, dále jen „MVC“, který se stará jak o výpočty a vyřizování požadavků, tak zobrazování výsledků uživateli. Obrázek 5 znázorňuje, jak MVC funguje v ASP.NET.

- **C#**
  Jako programovací jazyk jsem použil `C#`. Jsou v něm napsány všechny třídy a metody týkající se výpočtů a řízení aplikace. Je to objektově orientovaný jazyk.

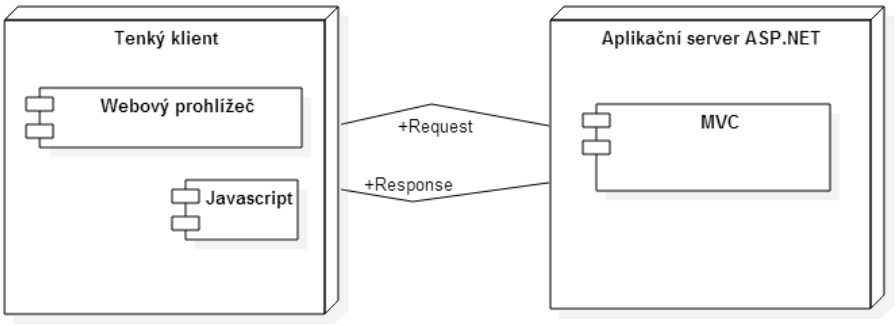
- **HTML**
  O zobrazení výstupu uživateli ve formě webových stránek se stará HTML (hyper text markup language).

- **Razor**
  Tato technologie je součástí ASP.NET a slouží ke generování dynamického obsahu do HTML. Jedná se o posílání dat z modelu a jejich reprezentaci uživateli. Je tak možné kombinovat používaný programovací jazyk a HTML.

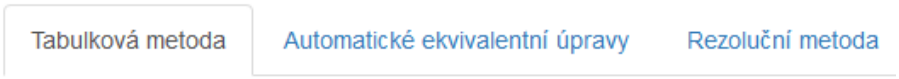
- **CSS**
  Kaskádové styly, dále jen „CSS“ jsou dnes součástí každého webu. Slouží k přehlednému stylyzování elementů na stránce. Obrovským pomocníkem se mi stala knihovna *Bootstrap*, která obsahuje předdefinované CSS styly. Její použití je snadné a velmi ulehčuje otázku designu. Při správném použití se stará o responzivní design a je tak možné výsledné stránky pohodlně používat na mobilních zařízeních, které disponují menší obrazovkou.

- **JavaScript**
  Tato technologie běží na klientském zařízení a hodí se všude tam, kde by bylo zbytečné používat server. V mé aplikace ji používám k obsluze uživatelských tlačítek k zadávání formulí.

Aplikace obecně funguje tak, že si uživatel ze zobrazených záložek (Obrázek 2) vybere, jaký chce typ úlohy. Na výběr je tabulková metoda, automatické ekvivalentní úpravy a rezoluční metoda. Dále zadá do vstupního pole formuli nebo více formulí (záleží jaký typ úlohy si vybral) a zvolí tlačítko k výpočtu. Následně se uživateli zobrazí kroky výpočtu. Pro každou úlohu se kroky výpočtu liší, podíváme se na ně blíže v kapitole 5.



Obrázek 1: Diagram nasazení



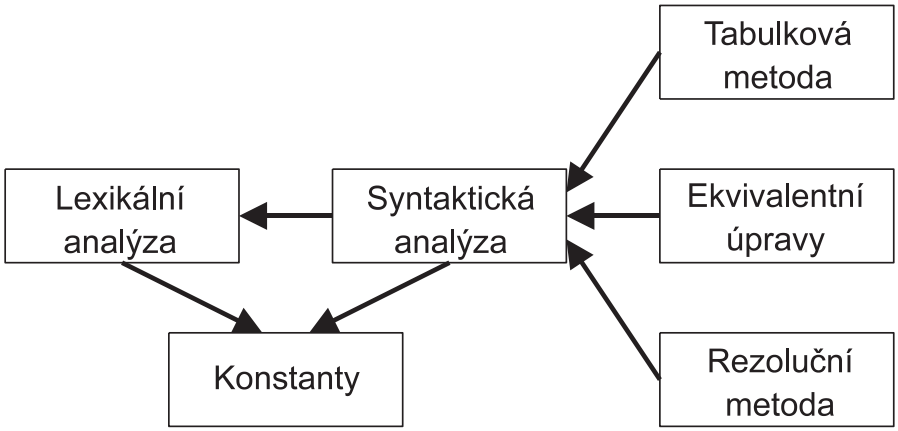
Obrázek 2: Ukázka vybrání typu úlohy

Celé formule může uživatel zadat přímo z klávesnice, nebo může využít připravených tlačítek, které slouží k zjednodušení zadávání především logických symbolů. Výstupy a kroky výpočtu jsou barevně stylyzovány k lepšímu znázornění nastalé situace. Aplikace počítá s tím, že uživatel může při zadávání formule chybovat, proto je po zadání nesprávné formule vypsána chybová hláška s povahou a místem chyby. U každého typu úlohy je k dispozici legenda s ovládáním aplikace a několik příkladů k otestování dané úlohy.

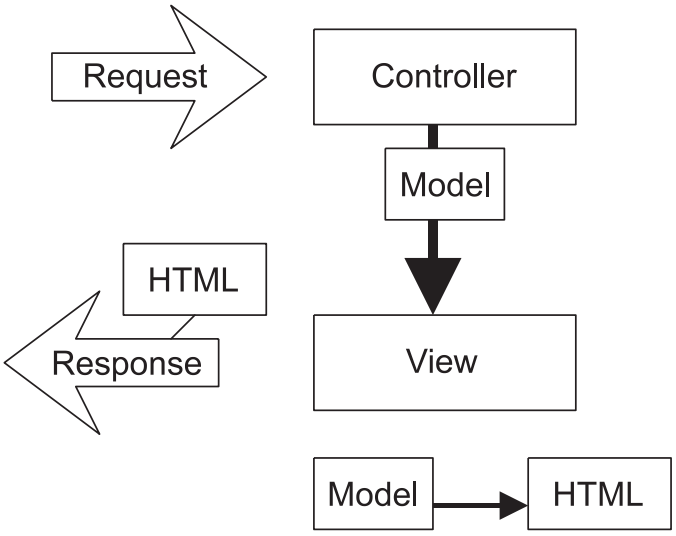
Obrázek 3: Ukázka zadávání formule

Aplikaci lze rozdělit na několik logických celků. Prvním z nich je zpracování uživatelského vstupu. Uživatel je vybízen pouze k zadání formulí. Tyto se musí zpracovat tak, aby s nimi bylo možné efektivně strojově pracovat. K tomu slouží lexikální a syntaktický analyzátor detailně popsaný v následující kapitole 4. Výsledkem těchto dvou analýz je derivační strom, reprezentující zadanou formuli. Dalšími logickými celky jsou už jednotlivé metody zpracování formulí. Tyto jsou popsány v kapitole 5. Obrázek 4 demonstruje propojení všech těchto celků, kde šipky vedoucí od jednoho celku ke druhému znamenají, že celek kam šipka směřuje, je využíván celkem, odkud

šipka vychází.



Obrázek 4: Diagram logických celků aplikace



Obrázek 5: Diagram zpracování požadavků MVC [5]

Zpracování požadavku podle Obrázku 5 probíhá tak, že je požadavek přijímut kontrolerem. Ten podle typu a povahy požadavku načte model, což znamená vytvoření objektu požadované třídy, a zavolá metodu této třídy, která má za úkol připravit všechna data, která mají být použita v pohledu. Pohled pak tato data zpracuje a odesílá odpověď ve formě HTML. Po zpracování požadavku jsou pak všechny objekty modelu zahozeny. Model by měl tedy správně obsahovat data, která budou zobrazována v pohledu a kontroler by měl obsahovat byznys logiku aplikace.

4 Zpracování uživatelských vstupů

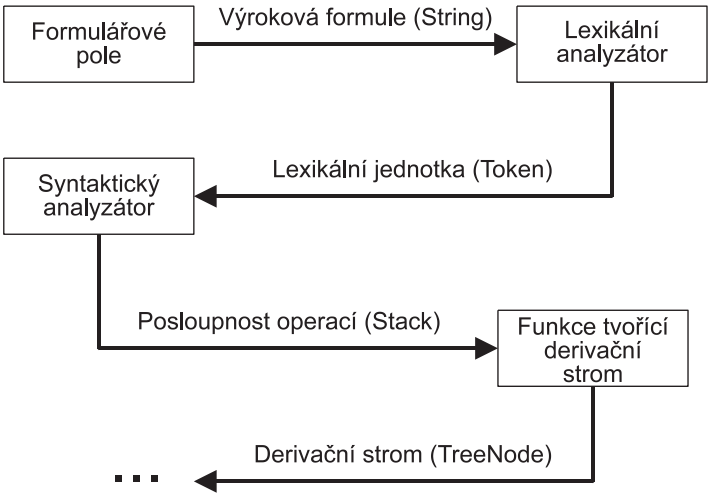
Uživatel zadává pouze formule. Zpracovává se tedy řetězec znaků a je třeba ho převést na strukturu, která mi určí pořadí operací a zařadí k nim správné identifikátory. V tomto případě jsou operacemi výrokové spojky a identifikátory výrokové symboly. K tomu je třeba dbát na závorky určující přednost operací. Celá tato struktura by pak měla jít zpětně převést na řetězec a mělo by se s ní dobře pracovat, ve smyslu úpravy jejího obsahu.

Jako nejlepší forma reprezentace takové formule a operací na ní se nabízí struktura stromu. Protože logické spojky výrokové logiky jsou až na unární negaci binární, bude strukturu formule tvořit binární strom. Abychom takový strom vytvořili, bude zapotřebí dvou analýz vstupní formule.

4.1 Lexikální analýza

První analýza, kterou nazýváme lexikální, má za úkol převést řetezec znaků zadaných uživatelem na tokeny. Token představuje objekt, který v našem případě nese informace o logických spojkách, symbolech a závorkách. U výrokových spojek je to informace pouze, o kterou spojku jde. U výrokových symbolů jde o dvě informace, že je to symbol a jaký má identifikátor. Výstupem lexikální analýzy tedy bude řada tokenů, uspořádaných tak, jak jdou za sebou ve formuli. Vynechají se tedy všechny mezery. Výrokové spojky, které jsou tvořeny více znaky, se stanou jedním tokenem.

Uživatelský vstup ale může obsahovat chyby. S tímto si musí lexikální analýza poradit a na neplatné znaky, či na neplatnou posloupnost znaků reagovat odpovídajícím způsobem. V mé aplikaci lexikální analyzátor vytvoří token, který říká, že došlo k chybě. Neodhalí však syntaktickou chybou, pouze zareaguje na neplatné znaky. V Tabulce 4 jsou vypsány všechny platné znaky nebo platné posloupnosti znaků.



Obrázek 6: Diagram transformace formule na derivační strom

mezera	tabulátor	(	)	&	∧	|	∨	~	¬	=>	⇒	<=>	⇔

Tabulka 4: Platné vstupy lexikální analýzy

```
//Vstup: retezec formule
//Vystup: jeden token
opakuj {
   typ tokenu := chybovy
   pokud (prvni znak formule == jedna z platnych operaci) pak {
      typ tokenu := prislusna operace
   }
   jinak pokud (prvni znak formule == identifikator) pak {
      typ tokenu := identifikator
      hodnota tokenu := hodnota identifkatoru
   }
   jinak pokud (prvni znak formule == mezera) pak {
      typ tokenu := mezera
      smaz prvni znak formule
   }
}
dokud (typ tokenu == mezera)
smaz prvni znak formule
vrat token
```

Výpis 1: Pseudokód metody `NextToken()` lexikálního analyzátoru

Ve Výpisu 1 je popsán pseudokód metody, která vrací právě jeden následující token. Je třeba říci, že tato metoda je součástí třídy lexikálního analyzátoru, která udržuje pomocný řetězec formule, ze kterého postupně odebírá znaky a převádí je na tokeny. Metoda `NextToken()` tedy při každém dalším spuštění pracuje s řetězcem formule, který je ochuzen o už zpracované části. Třída lexikálního analyzátoru je ovládána třídou syntaktického analyzátoru, který podle potřeby spouští metodu `NextToken()`. Pokud se při lexikálním zpracování nalezne chyba ve formuli, je pouze propagována syntaktickému analyzátoru.

4.2 Syntaktická analýza

Druhá analýza se zabývá syntaktickou skladbou tokenů a nazývá se proto syntaktická analýza. Pro svůj běh potřebuje dvě množiny informací. Jsou to gramatická pravidla a precedenční tabulka, která obsahuje informace o prioritě logických spojek. Díky těmto dvěma souborům pravidel a tokenům z lexikalní analýzy lze vytvořit strukturu nazývanou syntaktický strom. Tedy strukturu, která bude reprezentací formule uvnitř aplikace.

Syntaktická analýza využívá sadu gramatických pravidel obsažených v Tabulce 5. Tato pravidla definují, jak může být formule výrokové logiky syntakticky poskládána. Bezkontextová gramatika je definována jako uspořádaná čtveřice $G = (\Pi, \Sigma, S, P)$, kde

- Π je konečná množina neterminálních symbolů (neterminálů)
- Σ je konečná množina terminálních symbolů (terminálů), přičemž $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$ je počáteční (startovací) neterminál
- P je konečná množina pravidel typu $A \to \beta$, kde
  - A je neterminál, tedy $A \in \Pi$
  - β je řetězec složený z terminálů a neterminálů, tedy $\beta \in (\Pi \cup \Sigma)$

1	$\mathrm{F} \to \mathrm{i}$
2	$\mathrm{F} \to \neg\ \mathrm{F}$
3	$\mathrm{F} \to \mathrm{F} \wedge \mathrm{F}$
4	$\mathrm{F} \to \mathrm{F} \vee \mathrm{F}$
5	$\mathrm{F} \to \mathrm{F} \Rightarrow \mathrm{F}$
6	$\mathrm{F} \to \mathrm{F} \Leftrightarrow \mathrm{F}$
7	$\mathrm{F} \to (\mathrm{F})$

Tabulka 5: Bezkontextová gramatika pro formule výrokové logiky

Pravidla bezkontextové gramatiky jsou přepisovací pravidla, která nám říkají, jak můžeme přepsat výskyt neterminálu na levé straně pravidla, pravou stranou pravidla. Jedná se o skládání těchto pravidel do výsledné podoby nějakého výrazu takto definovaného jazyka. V našem případě můžeme z pravidel v Tabulce 5 odvodit jakoukoli formuli výrokové logiky.

Slovo „bezkontextová“ v názvu gramatiky znamená, že na levé straně každého pravidla stojí jeden neterminál bez sousedních symbolů. Tento neterminál můžeme při jakémkoli odvození přepsat podle uvedených pravidel nezávisle na jeho okolí, tedy bez ohledu na kontext.[6]

Gramatika definovaná podle Tabulky 5, je ale nejednoznačná. To znamená, že na jejím základě můžeme vytvořit více různých derivačních stromů (a z toho plyne i více různých syntaktických stromů) pro stejný řetězec. Toto je nežádoucí, protože požadujeme jeden derivační strom odpovídající prioritě logických spojek, která je popsána v kapitole 2. Buďto tedy mohu vytvořit jednoznačnou gramatiku, která pokryje priority logických spojek, nebo mohu postupovat jinak a tyto priority definovat pomocí precedenční tabulky. Rozhodl jsem se použít analyzátor, který využívá takovou precedenční tabulku a který se nazývá precedenční syntaktický analyzátor. Tento je často používán pro zpracování programovacích jazyků a je jednoduchý na implementaci. Konstrukci derivačního stromu provádí zdola-nahoru, což znamená, že ho konstruuje nahoru k jeho kořeni, tedy nalezne obrácenou posloupnost gramatických pravidel. Jinak řečeno vytváří pravý rozbor, což je reverzovaná posloupnost gramatických pravidel, která je použita v nejpravější

derivaci pro vstupní řetězec. U tohoto analyzátoru musí platit, že gramatika nemá více pravidel se stejnou pravou stranou a neobsahuje ϵ-pravidla.

4.2.1 Precedenční tabulka

Precedenční syntaktický analyzátor pracuje s pomocným zásobníkem, podle něhož a podle tokenů z lexikálního analyzátoru vybírá operaci z precedenční tabulky. Tedy podle toho, který znak je na vstupu a který symbol ja na vrcholu zásobníku, provede syntaktický analyzátor jednu ze čtyř operací $<$, $>$, $=$ nebo err [7]. Tyto operace jsou popsány v pseudokódu ve Výpisu 2. Pro lepší představu je v Tabulce 7 rozepsán postup, jak syntaktický analyzátor funguje uvnitř. Tabulka 7 obsahuje pohled do pomocného zásobníku, pohled na vstupní tokeny a výstup. Dále je potřeba si vysvětlit, co vyjadřuje precedenční Tabulka 6, která je použita v aplikaci.

Popis precedenční tabulky:

- První řádek tabulky představuje množinu **vstupních symbolů** - tokenů
- První sloupec obsahuje množinu **symbolů pomocného zásobníku**
- **$** je pomocný symbol
- ***i*** označuje identifikátor (výrokový symbol)
- Ostatní symboly jsou výše zmíněné operace $<$, $>$, $=$, **err**

Konstrukce precedenční tabulky:

Nechť $G = (\Pi, \Sigma, S, P)$, kde

- $\Pi = \{F\}$
- $\Sigma = \{(,), id_1, id_2, \ldots, id_m, op_1, op_2, \ldots, op_n\}$
- $S = \{F \to (F),\ F \to id_1,\ F \to id_2,\ \ldots,\ F \to id_m,$
  $F \to F\ op_1\ F,\ F \to F\ op_2\ F,\ \ldots,\ F \to F\ op_n\ F\}$
- $id_1, id_2, \ldots, id_m$ jsou identifikátory
- $op_1, op_2, \ldots, op_n$ jsou rozdílné operátory

Volba operací ($<$, $>$, $=$) pro precedenční tabulku:

- Precedence operátorů
  Pokud op_i má **vyšší prioritu** než op_j, potom:
  $op_i > op_j$ a $op_j < op_i$

- Asociativita operátorů
  Nechť op_i a op_j mají **stejnou prioritu**
  - Pokud op_i a op_j jsou **levě asociativní** potom:
    $op_i > op_j$ a $op_j > op_i$
  - Pokud op_i a op_j jsou **pravě asociativní** potom:
    $op_i < op_j$ a $op_j < op_i$

- Identifikátory
  Pokud $a \in \Sigma$ může být hned **před** id_i, pak: $a < id_i$
  Pokud $a \in \Sigma$ může být hned **za** id_i, pak: $id_i > a$

- Závorky
  Pro jeden pár závorek platí: $(=)$
  Nechť $a \in \Sigma - \{), \$\}$. Pak: $(< a$
  Nechť $a \in \Sigma - \{(, \$\}$. Pak: $a <)$
  Nechť $a \in \Sigma$ a a může být hned **před** (. Pak: $a < ($
  Nechť $a \in \Sigma$ a a může být hned **za**). Pak: $) > a$

- Ukončovač řetězce \$
  Nechť op_i je libovolný operátor, pak: $\$ < op_i$ a $op_i > \$$

- operace `err`
  Všude tam, kde v precedenční tabulce zůstalo volné pole, můžeme zapsat operaci `err`, která indikuje nepřípustnou kombinaci vstupních symbolů.

	¬	∧	∨	⇒	⇔	(	)	**i**	**$**
¬	<	>	>	>	>	<	>	<	>
∧	<	>	>	>	>	<	>	<	>
∨	<	<	>	>	>	<	>	<	>
⇒	<	<	<	>	>	<	>	<	>
⇔	<	<	<	<	>	<	>	<	>
(	<	<	<	<	<	<	=	<	err
)	err	>	>	>	>	err	>	err	>
i	err	>	>	>	>	err	>	err	>
$	<	<	<	<	<	<	err	<	err

Tabulka 6: Precedenční tabulka použitá v aplikaci

```
//Vstup: tokeny, bezkontextova gramatika, precedencni tabulka G = (Π, Σ, S, P)
    ; x ∈ Σ
//Vystup: pravy rozbor x, pokud x ∈ L(G), jinak chyba
//necht funkce Top() vraci terminal na zasobniku nejblize vrcholu
```

```
vloz "$" na zasobnik
opakuj {
   a := Top()
   b := NextToken()
   pokud (tabulka[a, b] == "=") pak {
      push(b)
      b := NextToken()
   }
   jinak pokud (tabulka[a, b] == "<") pak {
      zamen a za a"<" na zasobniku
      push(b)
      b := NextToken()
   }
   jinak pokud (tabulka[a, b] == ">") pak {
      pokud ("<"y je na vrcholu zasobniku a plati r : A → y ∈ S) pak {
         zamen "<"y za A
         vypis r na vystup
      }
      jinak chyba
   }
   jinak chyba
}
dokud (!(a == "$" && Top() == "$"))
```

Výpis 2: Pseudokód práce syntaktického analyzátoru

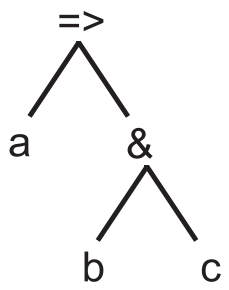
	zásobník	op	vstup	pravidlo
1	$	$<$	i $\Rightarrow$ i $\wedge$ i $	
2	$ $<$ i	$>$	$\Rightarrow$ i $\wedge$ i $	1. $F \to i$
3	$ F	$<$	$\Rightarrow$ i $\wedge$ i $	
4	$ $<$ F $\Rightarrow$	$<$	i $\wedge$ i $	
5	$ $<$ F $\Rightarrow$ $<$ i	$>$	$\wedge$ i $	1. $F \to i$
6	$ $<$ F $\Rightarrow$ F	$<$	$\wedge$ i $	
7	$ $<$ F $\Rightarrow$ $<$ F $\wedge$	$<$	i $	
8	$ $<$ F $\Rightarrow$ $<$ F $\wedge$ $<$ i	$>$	$	1. $F \to i$
9	$ $<$ F $\Rightarrow$ $<$ F $\wedge$	$>$	$	3. $F \to F \wedge F$
10	$ $<$ F $\Rightarrow$ F	$>$	$	5. $F \to F \Rightarrow F$
11	$ F		$	

Tabulka 7: Detailní příklad funkcionality syntantického analyzátoru pro vstup $i \Rightarrow i \wedge i\$$

Popis příkladu v Tabulce 7:

Ve výchozím stavu je na zásobníku pouze pomocný symbol $ a funkce `NextToken()` vrátila token reprezentující *identifikátor*. Podle této vstupní dvojice (vždy označené oranžovou barvou) vyhledal syntaktický analyzátor (dále jen SA) v precedenční tabulce znak $<$ představující operaci, která na zásobník vloží postupně znak $<$ a znak, který vrátila funkce `NextToken()` (v tomto kroku znak *identifikátoru*) a načte další token. Vše je připraveno k dalšímu kroku 2. SA zavolá metodu `Top()` (která vrací první neterminál ze zásobníku) a přečte token, který vrátila metoda `NextToken()`. Opět podle této kombinace vstupů nalezne příslušnou operaci v precedenční tabulce a vykoná její algoritmus. Pokud se provede operace $>$, znamená to, že se na základě obsahu zásobníku, použije některé z pravidel bezkontextové gramatiky a toto pravidlo se zapíše do jiného zásobníku, který slouží jako výstup SA. Podle tohoto výstupu už lze sestrojit derivační strom. Analýza končí buďto chybou, tzn. že se z precedenční tabulky načte jiný symbol než $<$, $>$, $=$, nebo na zásobníku a vstupu bude pomocný znak $.

Po dosazení výrokových symbolů a, b a c jako hodnot identifikátorů, vytvoří SA syntaktický strom jako je na Obrázku 7. Lze si všimnout, že operace $\wedge$ dostala přednost před operací $\Rightarrow$, tak jak je to nastaveno v precedenční tabulce podle přijaté konvence z kapitoly 2.



Obrázek 7: Syntaktický strom formule $a \Rightarrow b \wedge c$

4.3 Detekce chyb

Jak je naznačeno výše, aplikace umí detekovat jak lexikální, tak syntaktickou chybu. Výskyt lexikální chyby znamená, že se ve vstupní formuli nachází neplatný znak, nebo neplatná posloupnost znaků, které nelze převést na žádný token. Pokud k takové chybě dojde, SA obdrží chybový token a přestane ve vykonávání syntaktické analýzy. Zavolá metodu `makeErrorString()`, která nastaví příslušné chybové proměnné. Tyto obsahují informaci o tom, jaká chyba nastala a kde se tato chyba ve vstupní formuli nachází, aby se mohl neplatný znak zvýraznit v chybovém hlášení. Dále je chyba propagována výše do rodičovského objektu, který představuje jeden ze tří modulů, tak jak je zobrazeno na Obrázku 4. Příklad výpisu lexikální chyby, kde je místo výrokového symbolu napsáno číslo 8, lze vidět na Obrázku 8.

Podobně je tomu u chyby syntaktické povahy. Tu objeví SA, pouze když načte z precedenční tabulky pole `err`, což v našem případě znamená špatně utvořenou výrokovou formuli. SA využije

Obrázek 8: Příklad chybového výpisu s lexikální chybou

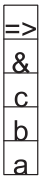
metodu lexikálního analyzátoru `makeErrorString()`, která poskytne informaci o místě výskytu chyby. Příklad výpisu syntaktické chyby, lze vidět na Obrázku 9.

Obrázek 9: Příklad chybového výpisu se syntaktickou chybou

5 Řešení konkrétních typů úloh

5.1 Implementace binárního stromu

Ještě než se dostaneme k jednotlivým typům úloh, je třeba uvést třídu aplikace `TreeOperations`. Tato třída má, kromě jiných věcí, za úkol z výstupu SA vytvořit binární syntaktický strom. Jak je zmíněno výše, výstupem SA je zásobník obsahující pravý rozbor formule. Strukturu zásobníku jsem použil proto, abych při jeho čtení dostal opačnou posloupnost výsledků. Tak mohu vytvořit strom od jeho kořene. Příklad obsahu zásobníku po syntaktické analýze formule $a \Rightarrow b \wedge c$ je zobrazen na Obrázku 10.



Obrázek 10: Příklad obsahu zásobníku při syntaktické analýze formule $a \Rightarrow b \wedge c$

K vytvoření syntaktického stromu jsem naimplementoval rekurzivní metodu `MakeTree()`. Tato metoda postupně vybírá operace ze zásobníku a podle těchto operací vytváří větve stromu. V našem příkladu vypadá vytvoření stromu následovně:

- Metodě je na začátku předán kořen stromu vytvořený z první položky zásobníku. Tímto kořenem tedy bude operace $\Rightarrow$.
- V prvním průchodu metodou se program dostane do větve s podmínkou, kde platí `node.data == "im"` (im = implikace).
- V této větvi vytvoří nový uzel stromu, což je uzel $\wedge$, který je pravým potomkem $\Rightarrow$ a pak na tohoto potomka zavolá tu stejnou metodu.
- Opět se vytvoří nový uzel stromu, který ale už bude patřit identifikátoru. Když se spustí další zanořená metoda pro identifikátor, program nevstoupí do žádné větve `if`, metoda tak skončí a program pokračuje v metodě o úroveň výše, kde se vytvoří levý potomek a spustí se motoda pro něho.
- Tímto způsobem se postupně vytvoří binární syntaktický strom jaký je na Obrázku 7.

```
// results.Pop() vrati a smaze prvni polozku zasobniku
// new TreeNode(results.Pop(), node) vytvori uzel stromu, kde prvni parametr
   konstruktoru je logicka spojka a druhy parametr je predek uzlu
private void MakeTree(TreeNode node)
{
```

```
    if (node.data == "(F)" || node.data == "not")
    {
        TreeNode child = new TreeNode(results.Pop(), node);
        node.AddChild(child);
        MakeTree(child);
    }
    else if (node.data == "or" || node.data == "and" || node.data == "im" ||
         node.data == "eq")
    {
        TreeNode child2 = new TreeNode(results.Pop(), node);
        node.AddChild(child2);
        MakeTree(child2);
        TreeNode child1 = new TreeNode(results.Pop(), node);
        node.AddChild(child1);
        MakeTree(child1);
    }
}
```

Výpis 3: Převedení výstupu SA na binární syntaktický strom

5.2 Implementace tabulkové metody

Vstupem tabulkové metody je jediná formule. Ta je zpracována způsobem popsaným v předchozích kapitolách, takže modul tabulkové metody pracuje na vstupu pouze s binárním syntaktickým stromem. Tento modul vytváří styl tabulky jako v příkladu v kapitole 2 v Tabulce 3. Dále podle výsledku určí, zda je formule tautologií, kontradikcí, či zda je splnitelnou. Jako poslední výstup modul vytváří ÚDNF a ÚKNF formule.

5.2.1 Výpočet tabulky

Nejprve se podívejme na tvorbu tabulky. V zásadě jde o určení pořadí v jakém se podformule (uzly syntaktického stromu) objeví v tabulce. Každý člověk si může tabulku poskládat různým způsobem, ale celkově by posloupnost podformulí měla mít postupně narůstající charakter. Určitě ale bude platit, že úplně vlevo v tabulce budou všechny výrokové symboly a úplně vpravo celá formule. Pro zbytek tabulky je třeba zavést pravidlo, podle kterého se určí posloupnost podformulí. Rozhodl jsem se, že pravidlo bude postupně vybírat uzly stromu od nejlevějšího po nejpravější a zdola nahoru. Pro lepší představu jsou na Obrázku 12 očíslovány uzly podle tohoto pravidla pro dříve zmíněnou Tabulku 3. Tímto pravidlem zajistím postupné narůstání podformulí a tabulka bude vypadat v naprosté většině případů přirozeně. Bylo myšleno i na situaci, kdy by došlo k tomu, že některá z podformulí by se už jednou v tabulce vyskytla. V

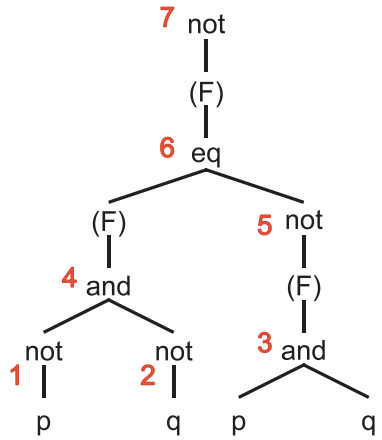
takovém případě by ji už člověk znovu nevypisoval. Stejně je tomu i v aplikaci, která kontroluje, zda již stejná podformule nebyla zpracována, jak lze vidět na Obrázku 11. Dále si lze všimnout, že ve výčtu operací byly vynechány závorky (F), ale pro lepší přehlednost ve výsledné tabulce jsou.

Vstup:

$$(a \Rightarrow b) \wedge (a \Rightarrow b)$$

a	b	(a ⇒ b)	(a ⇒ b) ∧ (a ⇒ b)
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1

Obrázek 11: Příklad vynechání stejné podformule



Obrázek 12: Pořadí operací v syntaktickém stromu formule $\neg((\neg p \wedge \neg q) \Leftrightarrow \neg(p \wedge q))$

K vypisování podformulí se používá metoda `TreeToFormula()`, která jako parametr přijme uzel stromu a jako výstup podá formuli vytvořenou od tohoto uzlu. Metoda `evaluateTree()` slouží k výpočtu pravdivostní funkce formule. Jako vstup používá jakýkoli uzel syntaktického stromu a seznam ohodnocených identifikátorů. Výstupem pak je pravdivostní ohodnocení uzlu. Další důležitou metodou je `fillResultTable()`, která na základě seznamu identifikátorů, seznamu operací a metody `evaluateTree()` sestaví výslednou tabulku.

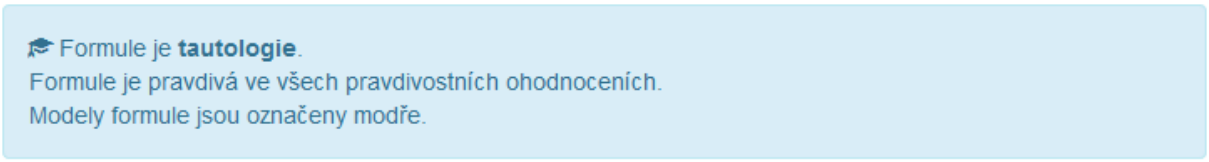
5.2.2 Určení typu formule

Každá dobře utvořená formule výrokové logiky je buďto splnitelná, tautologie nebo kontradikce. K určení jednoho z typů formule slouží metoda, která prochází výsledné hodnoty pravdivostních funkcí. Podle počtu těch, které jsou ohodnoceny jako „1“ a podle počtu všech hodnot, zobrazí výstup, viz. následující pseudokód 4.

```
urciTypFormule() {
   pocetPravdivychOhodnoceni := 0
   pro (vsechna ohodnoceni formule) {
      pokud (ohodnoceni formule == 1) {
         pocetPravdivychOhodnoceni++
      }
   }
   pokud (pocetPravdivychOhodnoceni == pocetVsechOhodnoceni) {
      typFormule := TAUTOLOGIE
   }
   jinak pokud (pocetPravdivychOhodnoceni == 0) {
      typFormule := KONTRADIKCE
   }
   jinak {
      typFormule := SPLNITELNA
   }
}
```

Výpis 4: Pseudokód určení typu formule

Aplikace výpíše o jaký typ formule se jedná a proč tomu tak je a ve výsledné tabulce označí modely formule. Příklad jak může vypadat výstup, když je formule tautologií je na Obrázku 13.



Obrázek 13: Ukázka výpisu typu formule

5.2.3 Vytvoření ÚDNF a ÚKNF

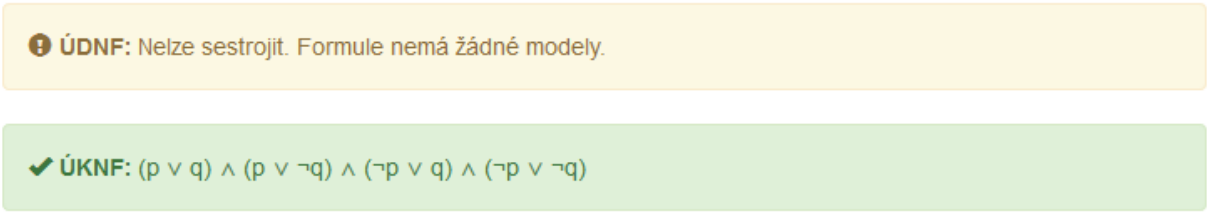
Obě normální formy jsou v aplikaci tvořeny stejným způsobem. Liší se pouze v tom, kdy je lze a kdy je nelze vytvořit a ve způsobu zápisu těchto forem.

Při tvoření ÚDNF jde obecně o to, poskládat správné ÚEK a ty pak dát do vzájemné disjunkce. Pokud již je tabulková metoda dokončena, nic nebrání tomu, využít jejich výsledků

a normální formy z nich vytvořit. Budou to právě modely formule, ze kterých lze vytvořit ÚEK. Jeden model bude sloužit k tvorbě jedné ÚEK. Naopak při vytváření ÚKNF a ÚED ,to nebudou modely, ale ta ohodnocení výrokových symbolů, která modely nejsou. Jak bylo popsáno v kapitole 2, pokud formule modely nemá, nelze vytvořit její ÚDNF. A zase opačně, pokud má formule pouze modely, nelze sestrojit její ÚKNF.

Model se na ÚEK převede tak, že pokud má identifkátor ohodnocení „1“, bude v ÚEK vystupovat prostě a pokud bude „0“, bude v negované podobě. U vytváření ÚED to bude přesně naopak. Pokud bude mít identifikátor pravdivostní ohodnocení „1“, bude v negované podobě a když bude mít ohodnocení „0“, bude v prosté podobě.

Aplikace počítá s tím, že v některých případech nelze vytvořit ÚDNF nebo ÚKNF a uživateli poskytne informace ve formě, jakou prezentuje následující Obrázek 14.



Obrázek 14: Ukázka výstupu pro ÚDNF a ÚKNF

Příklad celého výstupu modulu tabulkové metody pro formuli $\neg((\neg p \wedge \neg q) \Leftrightarrow (p \wedge q))$ je na Obrázku 15.

5.3 Implementace automatických ekvivalentních úprav

Tato část práce se zabývá nalezením posloupnosti takových ekvivalentních úprav, které vedou k vytvoření ÚDNF, DNF, ÚKNF a KNF. Lidé při používání ekvivalentních úprav pro převod do jedné z normálních forem mají mohou mít nějaký postup, kterým dojdou k vytvoření takové normální formy. Ovšem tohle nevždy platí a postupů, které vedou k cíli může existovat větší množství. Také si člověk v hlavě vytvoří jakýsi nadhled, díky kterému odhadne, jakou ekvivalentní úpravu dále použít, aby došel k zamýšlenému výsledku. Tento postup ale moc dobře nefunguje ve výpočetní technice. Hádání toho, jak dále postupovat na základě nějakých zkušeností a znalostí z dřívějška sice už v dnešní době je výpočetně možné, ale jak uvidíme dále, v tomto případě možná zbytečné.

Navíc, pokud má aplikace fungovat jako pomůcka při výuce, měla by jasně ukázat cestu, kterou se dostaneme k výsledku a posloupnost ekvivalentních úprav by měla mít nějakou logiku. Nakonec se mi takovou posloupnost podařilo najít. Jeví se mi jako kompromis optimální posloupnosti a komplexnosti algoritmu.

Forma v jaké uživatel zadá formuli, nemusí být taková jako ta, popsaná přijímanou konvencí v kapitole 2. Jde přednostně o závorkování. Priorita spojek je jasně daná precedenční tabulkou. Uživatel může použít libovolný počet závorek a nedopustit se tak syntaktické chyby. Z tohoto

Vstup:

$$\neg((\neg p \wedge \neg q) \Leftrightarrow \neg(p \wedge q))$$

p	q	¬p	¬q	(p ∧ q)	(¬p ∧ ¬q)	¬(p ∧ q)	((¬p ∧ ¬q) ⇔ ¬(p ∧ q))	¬((¬p ∧ ¬q) ⇔ ¬(p ∧ q))
0	0	1	1	0	1	1	1	0
0	1	1	0	0	0	1	0	1
1	0	0	1	0	0	1	0	1
1	1	0	0	1	0	0	1	0

Formule je **splnitelná**.
Existuje alespoň jedno pravdivostní ohodnocení, ve kterém je formule pravdivá.
Modely formule jsou označeny modře.

✔ **ÚDNF:** (¬p ∧ q) ∨ (p ∧ ¬q)

✔ **ÚKNF:** (p ∨ q) ∧ (¬p ∨ ¬q)

Obrázek 15: Ukázka výstupu modulu tabulkové metody

důvodu aplikace na začátku, než začne provádět ekvivalentní úpravy, může změnit podobu formule a závorky, které jsou navíc, odstranit. K tomu slouží několik metod, lišící se místem úpravy zbytečných závorek. Tak stejně se může stát, že po ekvivalentní úpravě provedené algoritmem vzniknou vzhledově nežádoucí formule. Opět se použijí tyto metody, které samozřejmě ctí syntaktickou skladbu formule a pracují výhradně se stromovou strukturou. Podobně tomu může být, když uživatel zadá několik negací za sebou. Syntakticky by podle analýzy formule $\neg\neg\neg a$ byla správná, ale na pohled a pro metody, které by s takovou formulí pracovaly, nepřípustná. Aplikace proto počet negací redukuje na nejmenší možný. Dokonce redukuje i kombinaci negací a závorek, kterou může představovat formule $((\neg(\neg(\neg(a)))))$. Po zpracování takové formule aplikace dále pracuje s její redukovanou verzí $\neg a$.

Když má aplikace nachystanou vstupní formuli, může na ni použít ekvivalentní úpravy. Všechny metody zahrnující ekvivalentní úpravy jsou z důvodu stromové struktury formule rekurzivní. Jejich základní podstatou je najít v derivačním stromu formule určité vzory a podle nich změnit strukturu stromu a aplikovat tak ekvivalentní úpravu. Například, když budu chtít použít De Morganův zákon pro konjunkci, budu ve formuli hledat negaci, která je použita na závorku, ve které je spojka $\wedge$. Metoda tedy nemá za úkol použít ekvivalentní úpravu na určité místo ve formuli, ale prohledává celou formuli a pokud najde místo, kde je možné uplatnit tuto úpravu, uplatní ji.

Tímto způsobem můžu například použít metodu uplatňující De Morganův zákon, abych mohl použít metodu pro distributivní zákon a následně po distribuci nalézt vzniklé tautologie nebo kontradikce a s nimi zase podle jiných zákonů nějak naložit. V diagramu na Obrázku 19 lze vidět posloupnost těchto metod, díky které se formule vždy upraví do podoby DNF či KNF a z této formy pak do ÚDNF nebo ÚKNF.

Když některá z metod změní strukturu formule a aplikuje se tak některá z ekvivalentních úprav, přídá se tato úprava jako další krok řešení a tyto kroky dohromady tvoří výstup modulu ekvivalentních úprav. Převod do normálních forem může skončit různými způsoby. Pokud převádím formuli do ÚNDF a zjistí se po všech ekvivalentních úpravách, že formule je kontradikcí, jako poslední krok výstupu aplikace upozorní na tuto skutečnost. Pokud bude formule tautologií, posledním krokem převodu bude zobrazení úplné normální formy skládající se ze všech ÚEK. Pro ÚKNF bude situace přesně opačná. To co u ÚDNF platilo pro tautologii bude platit u ÚKNF pro kontradikci a opačně s tím rozdílem, že pokud bude formule kontradikcí bude se výsledná forma skládat ze všech ÚED. Poslední možnou situací je, že po všech ekvivalentních úpravách nebude formule ani tautologií, ani kontradikcí, ale bude převedná do DNF nebo KNF a z této formy pak rošířena do ÚDNF nebo ÚKNF. Příklady těchto možných ukončení jsou na Obrázcích 16, 17 a 18.

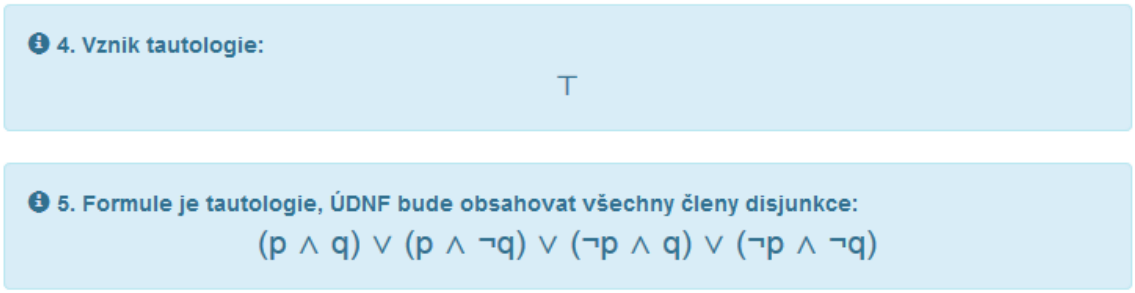
Popis metod v diagramu na Obrázku 19

- **Úprava závorek**
  Tento krok zahrnuje několik metod. Jde o odstranění závorek obklopující celou for-

muli, odstranění vícenásobného závorkování podformulí a přidání závorek tam, kde je potřeba zdůraznit prioritu operátorů. Např. se formule $a \vee b \wedge c$ změní na formuli $a \vee (b \wedge c)$.

- **Odstranění ekvivalence a implikace**
  Aby bylo možné aplikovat De Morganovy zákony a jiné zákony, které používají pouze logické spojky $\wedge$ a $\vee$, je potřeba formuli zbavit ekvivalencí a implikací. Popis metody odstranění implikace lze nalézt níže v textu.
- **Použití De Morganových zákonů**
  Aby bylo možné aplikovat distribuční zákony, je třeba ve formuli odstranit negace před závorkama. Cílem této metody je přesunout pomocí De Morganových zákonů všechny negace ve formuli k výrokovým symbolům.
- **Nalezení tautologií a kontradikcí**
  Při vykonávání metod ekvivalentních úprav může v této chvíli dojít k situaci, kdy mohou vznikat formule jako $a \vee \neg a$, $a \wedge \neg a$. Aplikace vyhodnotí tyto formule jako tautologie a kontradikce a nahradí je příslušnými symboly $\top$ a $\bot$. Tímto zásahem do formule, ale mohou vzniknout formule jako $a \wedge \top$, $a \wedge \bot$, $a \vee \top$ a $a \vee \bot$. Podle zákonů zmíněných v kapitole 2 dojde buď k tomu, že tautologie a kontradikce nebudou mít vliv na výslednou formuli a můžou se odstranit, nebo dojde k jejich agresivnosti a celá výsledná formule se stane tautologií nebo kontradikcí.
- **Použití distributivního zákonu**
  Pokud chci formuli převést do jedné z disjunktivních normálních forem, je vhodné použít distributivní zákon tak, aby tvořil elementární konjunkce a pro konjunktivní formy, aby tvořil elementární disjunkce. Aplikace tedy podle toho do jaké formy převádím formuli, používá jen jeden ze zákonů.
  Pro ÚDNF a DNF zákon $\models (\mathbf{p} \vee \mathbf{q}) \wedge \mathbf{r} \Leftrightarrow (\mathbf{p} \wedge \mathbf{r}) \vee (\mathbf{q} \wedge \mathbf{r})$
  Pro ÚKNF a KNF zákon $\models (\mathbf{p} \wedge \mathbf{q}) \vee \mathbf{r} \Leftrightarrow (\mathbf{p} \vee \mathbf{r}) \wedge (\mathbf{q} \vee \mathbf{r})$.
- **Použití asociativního zákonu**
  V tuto chvíli se již formule nachází v DNF (KNF), jen je třeba pomocí asociativního zákonu odstranit závorky v elementárních konjunkcích (disjunkcích), jednak aby formule splňovala vzhled normální formy a jednak aby se připravila její struktura na další metody. Ukázka pro lepší pochopení se nachází na Obrázku 20.
- **Odstranění stejných identifikátorů**
  Jak lze vidět na Obrázku 20, ve formuli se nachází přebytečné výrokové symboly a ty je třeba podle zákonu idempotence odstranit. Výsledek lze vidět na Obrázku 21.
- **Nalezení tautologií a kontradikcí podruhé**
  Po užití distributivního zákonu může opět nastat situace, kdy se ve formuli objeví tautologie či kontradikce.

- **Rozšiřování členů DNF (KNF)**
  V tomto kroku je důležité si uvědomit, že pokud převádím formuli do úplné normální formy, mohou se některé identifikátory po aplikaci ekvivalentních úprav vytratit. Je proto nutné najít v původní formuli všechny identifikátory a při rozšiřování je použít.
- **Výsledná formule v ÚDNF (ÚKNF)**
  Po rozšíření se mouhou vyskytnout složením stejné ÚEK (ÚED). Ty je třeba odstranit. Ve zbylých ÚEK (ÚED) aplikace seřadí výrokové symboly podle abecedy.

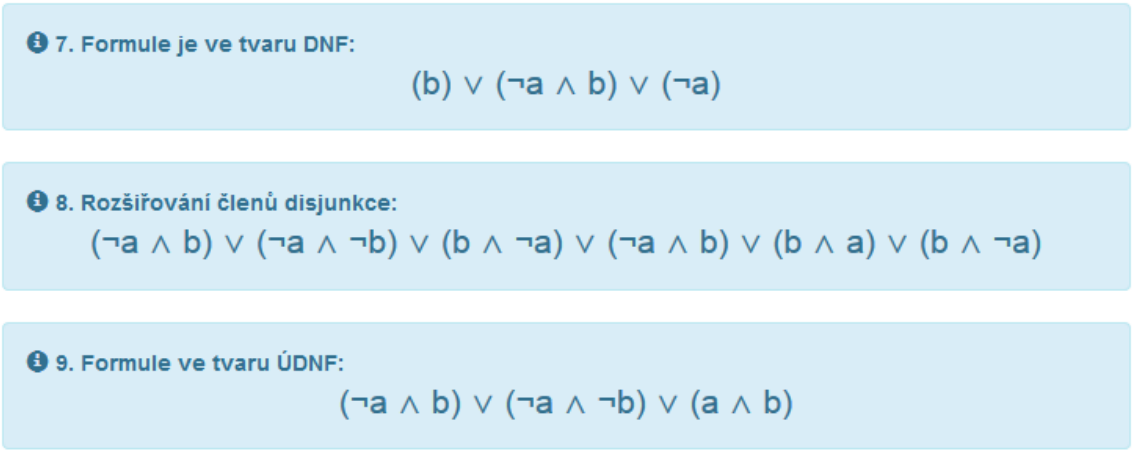


Obrázek 16: Příklad posledního kroku převodu do ÚDNF, když je formule tautologie

Obrázek 17: Příklad posledního kroku převodu do ÚKNF, když je formule tautologie

---

```
public void im2or(TreeNode node)
{
   if (isFound) return;
   if (node.data == "im")
   {
      node.newData("or");
      TreeNode left = node.getChild1();
      TreeNode neg = new TreeNode("not", node);
      TreeNode br = new TreeNode("(F)", neg);
      node.newChild1(neg);
      neg.AddChild(br);
      br.AddChild(left);
```



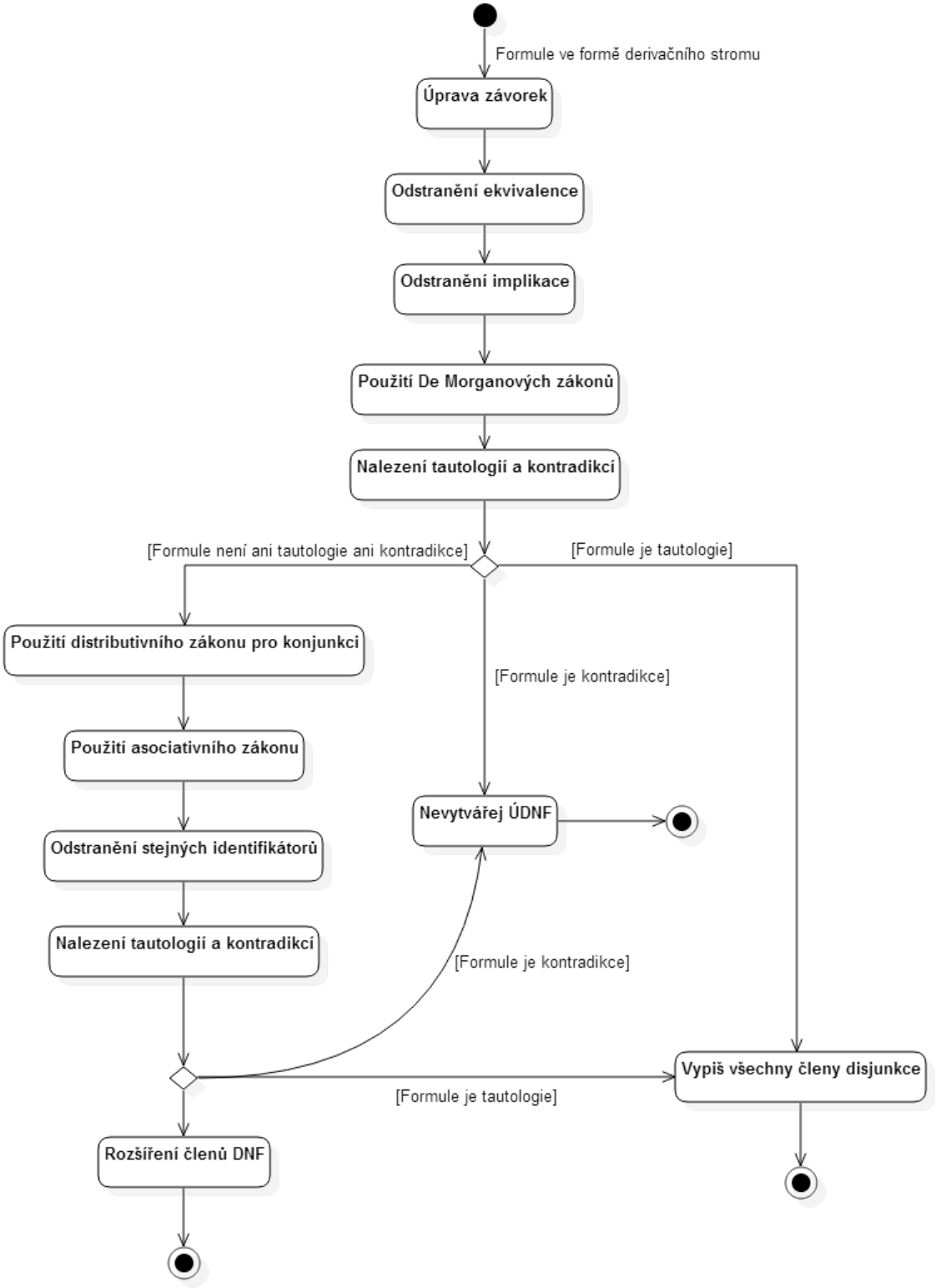
Obrázek 18: Příklad posledního kroku převodu do ÚDNF, když formule není ani tautologie, ani kontradikce

```
      left.newParent(br);
      isFound = true;
   }
   else if (node.data == "or" || node.data == "and" || node.data == "eq")
   {
      im2or(node.getChild1());
      im2or(node.getChild2());
   }
   else if (node.data == "(F)" || node.data == "not")
   {
      im2or(node.getChild2());
   }
}
```

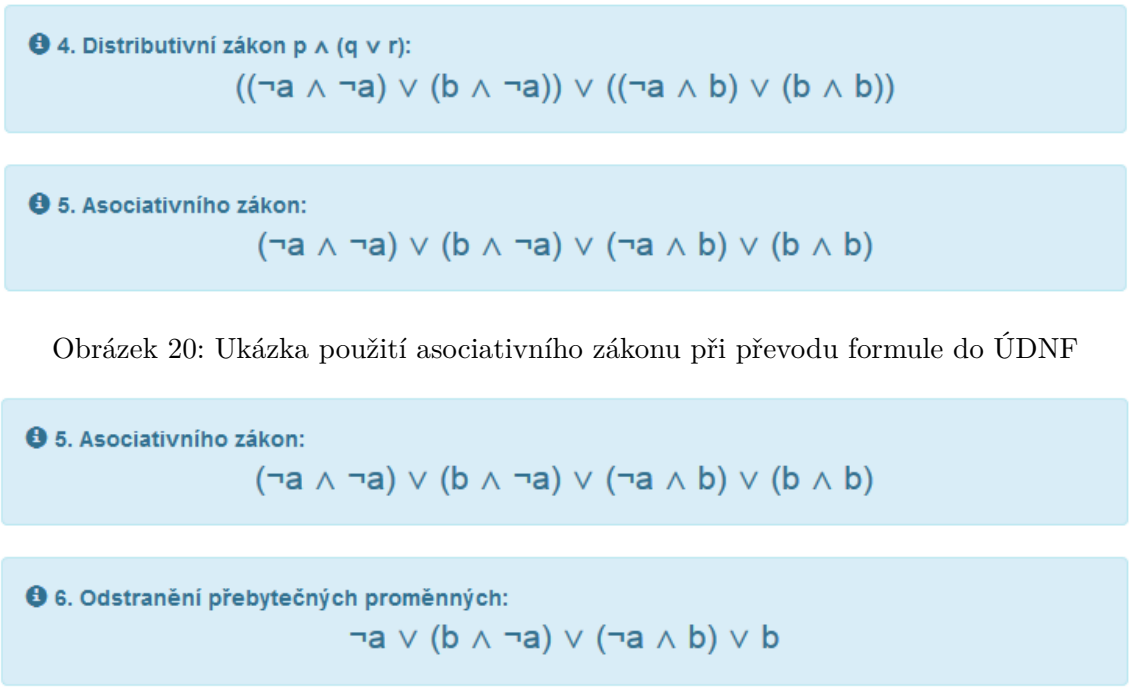
Výpis 5: Metoda `im2or()` odstraňují implikaci

Většina metod v modulu ekvivalentních úprav má podobný charakter. Hledají v syntaktickém stromu určitý vzor a pokud ho najdou, provedou změnu ve struktuře stromu na základě jednoho ze zákonů pro ekvivalentní úpravy. Protože těchto metod je v aplikaci implementován větší počet, popíši funkčnost pouze jedné z nich, za účelem demonstrovat obecnou funkcionalitu všech. Jedná se o metodu ve Výpisu kódu 5, která odstraňuje implikaci na základě zákonu $(\mathbf{p} \Rightarrow \mathbf{q}) \Leftrightarrow (\neg\mathbf{p} \vee \mathbf{q})$.

Vstupním parametrem metody je uzel syntaktického stromu, od kterého začít hledat. Při prvním spuštění bude parametrem kořen stromu. Proměnná `isFound` slouží k identifikování toho, zda metoda vykonala změnu v syntaktickém stromu. Pokud tato proměnná nabyje hodnoty `true`, tedy došlo ke změně, nebude se dál metoda vykonávat a tato změna se uloží jako jedna



Obrázek 19: Diagram popisující převod formule do ÚDNF



Obrázek 20: Ukázka použití asociativního zákonu při převodu formule do ÚDNF

Obrázek 21: Ukázka odstranění přebytečných identifikátorů

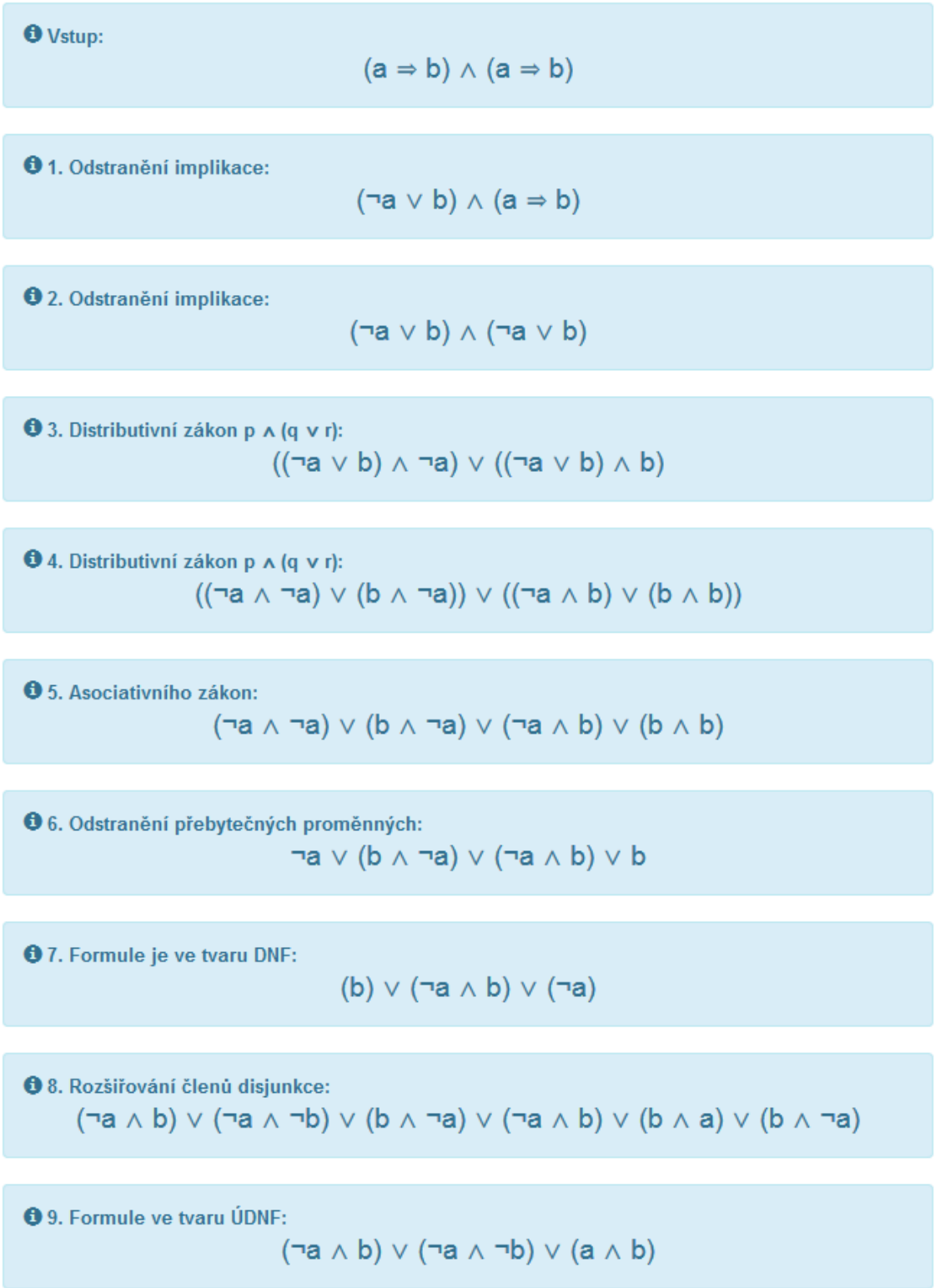
položka seznamu ekvivalentních úprav. Popisovaná metoda je pak spuštěna znovu od kořene stromu a hledá další výskyt implikace. Pokud nedojde k nálezu a nezmění se tak struktura stromu, metoda již nebude volána.

Pokud dojde k nalezení implikace, okamžitě nastává změna struktury. Nejprve se uzel implikace změní na disjunkci. Nyní je potřeba upravit levou stranu bývalé implikace na její negaci. Vytvoří se dva nové uzly. Jeden bude představovat negaci a druhý uzávorkování této negace. Předkem uzlu negace se stane nynější uzel disjunkce a předkem uzlu představující závorky se stane uzel negace. Je třeba nastavit i nového předka pro původní levou stranu implikace, kterým se stane uzel uzávorkování (F). Dále je nutné nastavit ukazatele na potomky nejen pro nově vytvořenou negaci a závorky, ale i pro nynější disjunkci, čímž bude nově vzniklá negace. Při takovémto vytváření vazeb v derivačním stromu je velmi snadné udělat chybu, nebo opomenout některou z vazeb. Především v komplikovanějších situacích, než je tato popisovaná.

Jak metoda hledá uzel, kterým je v tomto případě implikace, zanořuje se rekurzivně hlouběji do stromu a volá samu sebe s novým parametrem, kterým je potomek prohledávaného uzlu. Pokud tedy narazí na uzel, jež má mít dva potomky, musí volat samu sebe na oba dva a tak projít celou strukturu stromu.

5.4 Implementace rezoluční metody

Rezoluční metodou můžeme řešit více typů úloh. V aplikaci lze řešit tři typy úloh, které jsou popsané v kapitole 2.



Obrázek 22: Příklad výstupu výpočtu ÚDNF pro formuli $(a \Rightarrow b) \wedge (a \Rightarrow b)$

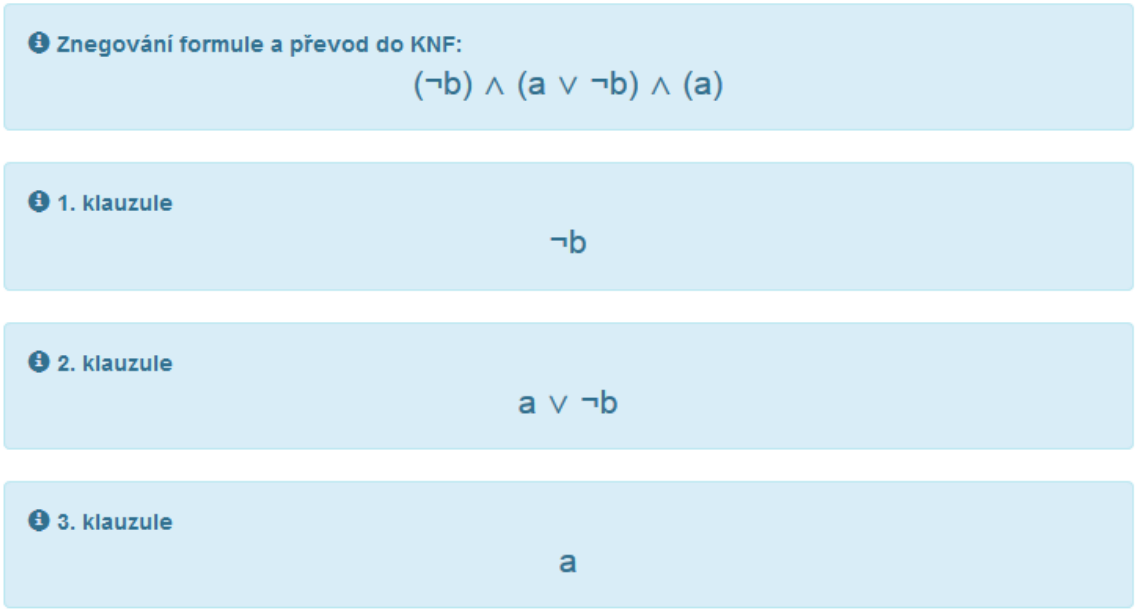
5.4.1 Důkaz tautologičnosti formule

První z úloh je provést důkaz, že zadaná formule je tautologie. V tomto případě je vstupem pouze jediná formule. Tu je potřeba znegovat a převést do KNF. Negaci aplikace provede tak, že obalí řetězec vstupní formule do $\neg()$. Tento vstupní řetězec pak předá standardně lexikální a syntaktické analýze. Pokud analýzy proběhnou v pořádku, proběhne převod formule do KNF. V tomto bodě nastává problém, protože při převodu do KNF mohou nastat dvě nepříznivé situace, kdy převod končí výsledkem, že je formule kontradikce nebo tautologie. Algoritmu pro rezoluci se tak nedostane klauzulí, se kterými by mohl pracovat. Vstupní formuli, která bude opravdu tautologií a po znegování se stane kontradikcí, tak nebude moci rezoluční algoritmus vůbec zpracovat. V tomto případě bude výsledek rezoluce pouze zpráva o tom, že zadná formule je tautologie. Podobná situace nastane, když bude vstupní formule kontradikce a po znegování se stane tautologií. Rezoluční algoritmus opět nedostane požadované klauzule jako vstupy a pouze se vypíše zpráva o tom, že zadaná formule je kontradikce. Situace, kdy se formule stane tautologií nebo kontradikcí, nastane pouze pokud dojde v ekvivalentních úpravách k aplikaci zákonu $\models \mathbf{p} \vee \neg\mathbf{p}$ nebo $(\mathbf{p} \wedge \neg\mathbf{p}) \Leftrightarrow \mathbf{0}$. Jen tehdy algoritmus rozpozná, že jde o tautologii nebo kontradikci, což znamená, že některé formule, ač jsou tautologie nebo kontradikce, mohou být zpracovány dále pomocí rezolučního algoritmu. Například formule $\neg a \wedge (b \Rightarrow a) \wedge b$ se pomocí algoritmu převádějící formuli do KNF v modulu ekvivalentních úprav nestane kontradikcí, i když jí je. Jediná možnost, kdy rezoluční algoritmus může běžet, je, když algoritmus převádějící formuli do KNF neurčí, že je formule tautologie nebo kontradikce.

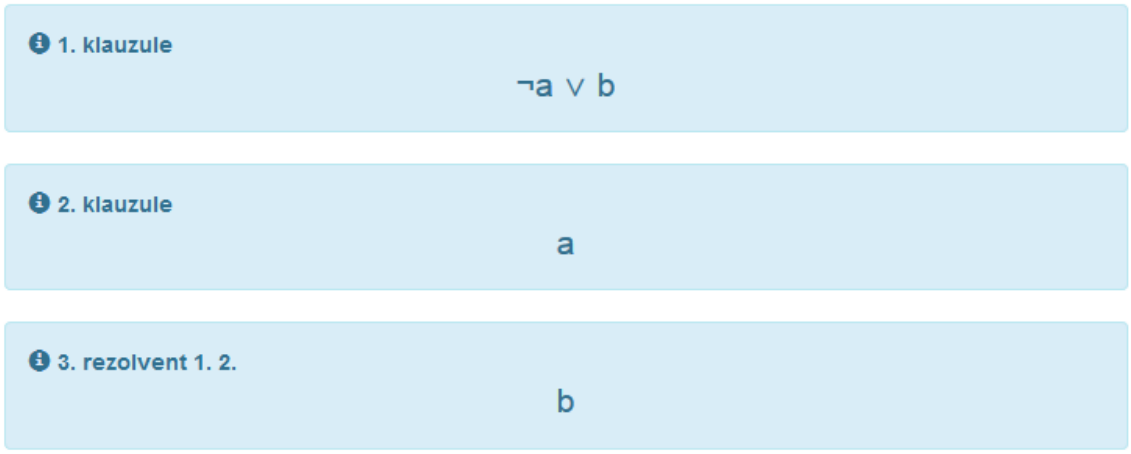
Pokud má rezoluční algoritmus na vstupu formuli v KNF, rozdělí ji na jednotlivé klauzule. K tomu je využita metoda, která prochází syntaktický strom a hledá jednotlivé elementární disjunkce. Tato metoda vytvoří speciální strukturu do které je klauzule uložena. Na Obrázku 23, lze vidět příklad rozdělení formule na klauzule.

Nyní je možné hledat v klauzulích literály, na které lze uplatnit rezoluční pravidlo. Algoritmus porovnává klauzule mezi sebou a v nich porovnává literály. Pokud najde v jedné klauzuli výrokový symbol v prosté podobě a v druhé klauzuli ten stejný symbol v negované podobě, odstraní tyto literály z klauzulí a sloučí klauzule dohromady. Pak se algoritmus podívá do seznamu klauzulí a takto vytvořených rezolventů a pokud se teď vytvořený rezolvent v seznamu nenajde, přidá ho jako nový. Ještě než se rezolvent porovná s ostatníma klauzulema a rezolventama v seznamu, seřadí se v něm literály podle abecedy. Vytvoření rezolventu lze vidět na Obrázku 24.

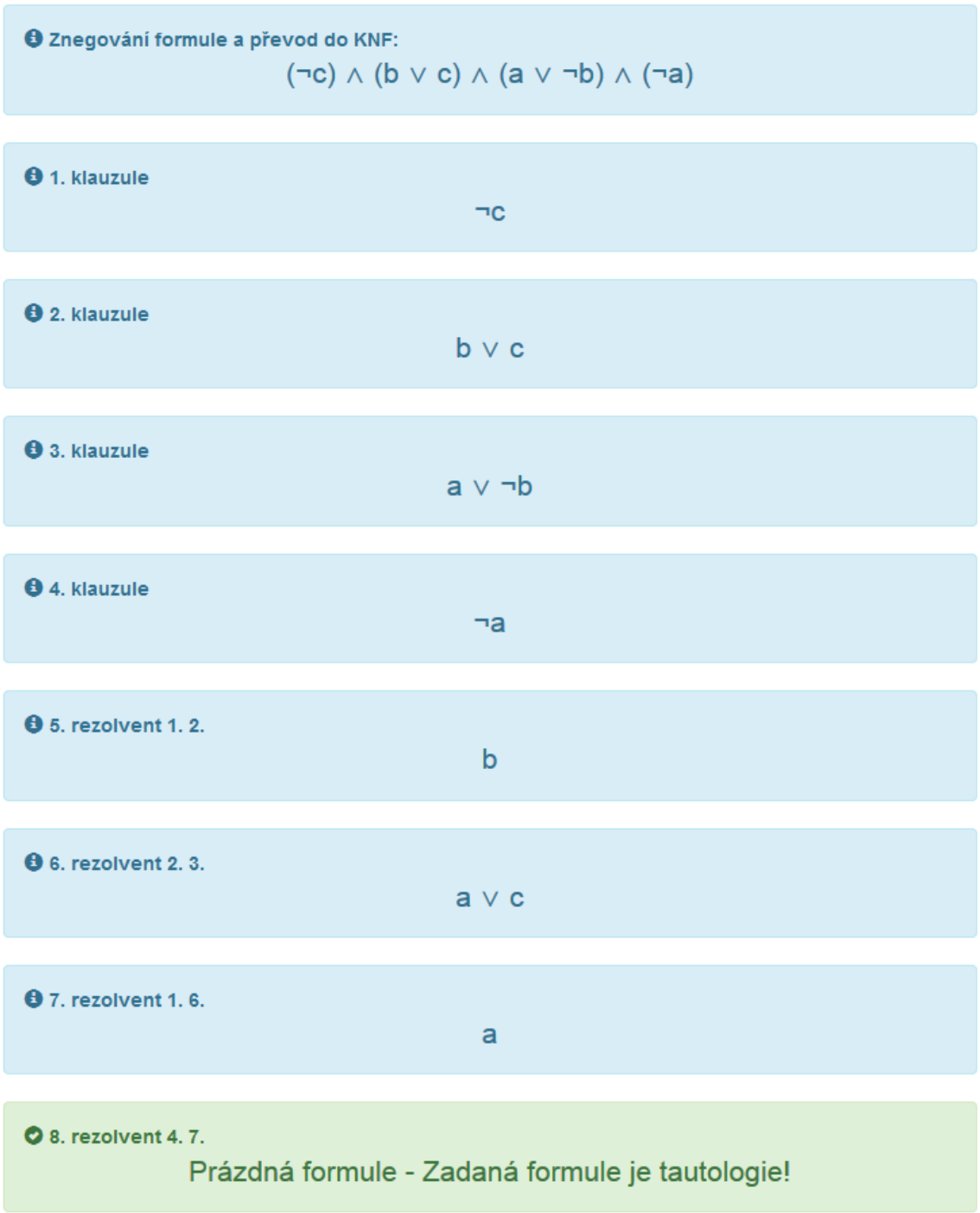
Aplikace vypíše rezolvent a k němu z jakých klauzulí či rezolventů vznikl. Pokud algoritmus už nenajde místo, kde uplatnit rezoluční pravidlo, výpočet končí a aplikace vypíše zprávu o neúspěchu nalezení prázdné formule. Pokud se prázdná formule najde a je zvolena úloha provedení důkazu tautologičnosti formule, vypíše se zpráva, že formule je tautologí. Příklad lze vidět na Obrázku 25.



Obrázek 23: Příklad rozdělení formule na klauzule



Obrázek 24: Příklad vytoření rezolventu



Obrázek 25: Příklad důkazu tautologičnosti formule $\neg(\neg a \wedge (b \Rightarrow a) \wedge (\neg c \Rightarrow b) \wedge \neg c)$

5.4.2 Důkaz nesplnitelnosti množiny klauzulí

Dalším typem úlohy s využitím rezoluční metody je důkaz nesplnitelnosti množiny klauzulí. Nyní je po uživateli aplikace požadováno, aby zadal několik klauzulí. Klauzule musí být zadány správně, tedy musí být ve formě elementárních disjunkcí, jinak aplikace vypíše varovnou zprávu a algoritmus rezoluce se nespustí. Zadávání klauzulí si lze prohlédnout na Obrázku 26. Přidávání a odebírání klauzulí je efektivně řešeno pomocí tlačítek plus a minus.

Když jsou klauzule zkontrolovány a uznány za platné, jsou spojeny do jedné formule a algoritmus využije metody na vyhledávání klauzulí ve formuli a převede tak klauzule do struktury potřebné pro další zpracování. Nyní algoritmus postupuje stejně jako v předchozím případě a používá rezoluční pravidlo k nalezení prázdné formule. Rozdíl nastane až při výpisu výsledku. Pokud algoritmus nalezne prázdnou formuli, vypíše zprávu, že zadaná množina klauzulí je nesplnitelná. Pokud nenajde prázdnou formuli, množina formulí není nesplnitelná.

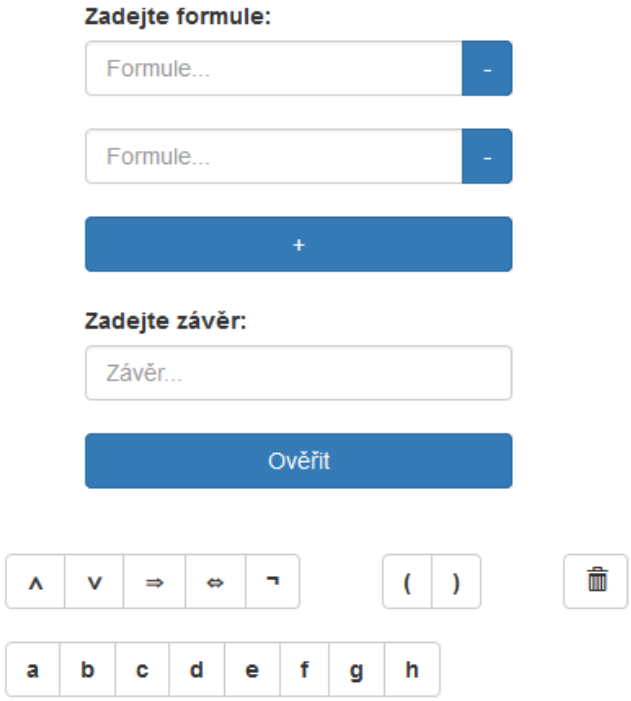
Obrázek 26: Způsob zadávání vstupů k úloze nesplnitelnosti množiny klauzulí

5.4.3 Důkaz správnosti úsudku

Posledním typem úlohy používající rezoluční metodu je důkaz správnosti úsudku. Po uživateli je požadováno zadat formule jako předpoklady a jednu formuli jako závěr, viz. Obrázek 27. Je potřeba zjistit, zda daný závěr logicky vyplývá z předpokladů. To lze zjistit tak, že formule, které jsou zadány jako předpoklady převedeme na klauzule. Závěr, který znegujeme k těmto klauzulím přidáme. Máme tedy stejně jako v minulé úloze množinu klauzulí a ptáme se, zda je tato množina nesplnitelná. Pokud je nesplnitelná, znamená to, že úsudek je platný.

Tohle bude platit ale pouze tehdy, když bude množina klauzulí konzistentní. To znamená, že konjunkce všech klauzulí předpokladů nesmí být kontradikce. Pokud by kontradikcí byla, může

z těchto předpokladů vyplývat jakýkoli závěr. Aplikace tedy prvně zkontroluje předpoklady a až potom řeší nesplnitelnost množiny klauzulí. Rozdíl oproti minulé úloze je také v tom, že se nemusí zadávat pouze klauzule, ale lze zadat jakékoli formule a to jak do vstupních polí předpokladů, tak do vstupního pole závěru.



Obrázek 27: Způsob zadávání vstupů k úloze ověření správnosti úsudku

Každá formule zadaná jako předpoklad je převedena do KNF stejným způsobem jako v důkazu tautologičnosti formule. Mohou tedy opět nastat případy, kdy po převodu do KNF bude formule vyhodnocena jako tautologie nebo kontradikce. V tomto typu úlohy jsou takové výsledky užitečné. Pokud by jedna z formulí byla tautologie, pouze ji vypustím z výpočtu. Pokud by se zjistilo, že je některá formule kontradikce, předpoklady by tak byly nekonzistentní a rezoluce by neproběhla [8].

6 Závěr

Při práci jsem největší pozornost věnoval ekvivalentním úpravám použitých k převodu formulí do normálních forem. Tato část byla implementačně nejsložitější a časově nejnáročnější. Co se týče výsledků těchto převodů, je zde prostor pro vylepšování. Zejména při použití distributivního zákonu, kde postupné kroky vedou u formulí s více výrokovými symboly k obrovským formulím, což může u uživatele aplikace budit dojem složitosti a nepřehlednosti.

Velký důraz byl kladen na vzhled formulí a splnění přijímané konvence o zápisu formulí. Jakákoli formule vytvořená aplikací dodržuje tato pravidla. Mnoho snahy jsem vynaložil také k tomu, aby uživatel mohl zadat jakýkoli vstup a aplikace si s ním uměla poradit. Ať už se jedná o několikanásobné použití závorek nebo negací a jiných podivných konstrukcí, či naopak prázdných formulí, aplikace by si s tímto měla poradit. Aplikace byla odzkoušena na mnoha formulích, ale i tak se může stát, že jsem některý z detailů opomněl a dojde k neočekávané chybě.

Vylepšením aplikace by se mohla stát podpora více typů závorek ke zvýšení přehlednosti jak generovaných formulí, tak těch, které může zadávat uživatel. Dalším vylepšením by se mohlo stát vyřešení problému s převodem formulí do KNF pro typ úlohy hledání důkazu tautologičnosti formule pomocí rezoluční metody tak, aby z formulí nevznikaly tautologie a kontradikce, které již dále nelze k rezoluci použít. Tedy aby vždy vznikla formule v KNF. U tabulkové metody by mohla vzniknout možnost vytvoření i druhého popisovaného typu tabulky v kapitole 2.

Uživatelské rozhraní aplikace a zobrazování výsledků jsem se snažil udělat přehledné, názorné a co nejjednodušší. Zde byla výše zmiňovaná CSS knihovna Bootstrap obrovským přínosem. Ovládání aplikace by mělo být zcela intuitivní a zadávání formule, díky tlačítkům s logickými spojkami, rychlé.

Aplikace je dostupná pro kohokoli na webové adrese http://vl-vsb.aspone.cz/.

Literatura

[1] VELEBIL, Jiří. *Karnaughovy mapy* [online]. Praha: FEL ČVUT, 2000, 18 s. [cit. 2016-04-25]. Dostupné z: <http://users.fit.cvut.cz/ staryja2/BIMLO/readme/velebil-karnaughovy-mapy.pdf >.

[2] DUŽÍ, Marie. *Logika pro informatiky* Vyd. 1. Ostrava: Ediční středisko VŠB-TUO, 2012, 183 s. ISBN 978-80-248-2662-2.

[3] ROBINSON, J. A. *A Machine-Oriented Logic Based on the Resolution Principle* New York, NY, USA: Journal of the ACM, 1965, vol. 12, iss. 1, s. 23-41. ISSN: 0004-5411.

[4] HROMEK, Petr. *Logika v příkladech* Vyd. 1. Olomouc: Univerzita Palackého v Olomouci, 2002, 211 s. ISBN 80-244-0578-4.

[5] GALLOWAY, Jon. *Introduction to ASP.NET MVC* [online]. last revision 9th of August 2014 [cit. 2016-04-25]. Dostupné z: <https://mva.microsoft.com/en-US/training-courses/introduction-to-asp-net-mvc-8322 >.

[6] JANČAR, Petr. *Teoretická informatika - učební text* Vyd. 1. Ostrava: Ediční středisko VŠB-TUO, 2007, 330 s. ISBN 978-80-248-1487-2.

[7] MEDUNA, Alexander. *Syntaktická analýza zdola nahoru* [online]. last revision 12th of September 2015 [cit. 2016-04-25]. Dostupné z: <https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj08-anim-cz.pdf >.

[8] SAWA, Zdeněk. *Logické vyplývání* [online]. last revision 18th of February 2016 [cit. 2016-04-25]. Dostupné z: <http://www.cs.vsb.cz/sawa/uti/slides/uti-03-cz.pdf >.

Seznam příloh

A Příloha na CD

Kód celého programu vytvořeného pro tuto práci se nachází na přiloženém CD v adresáři `Program`.